НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра автоматики та управління в технічних системах

«На правах рукопису»
УДК _____

До захисту допущено:
Завідувач кафедри
_____ Олександр, РОЛІК
«___»_____2021р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-науковою програмою «Інженерія програмного забезпечення комп'ютерних систем»

зі спеціальності 121 «Інженерія програмного забезпечення»

на тему: «Організація програмних систем надання сервісів в музичній галузі»

Виконав:
студент VI курсу, групи ІТ-91мн
Поліщук Мирослав Ігорович _____

Керівник:
Професор кафедри АУТС, д.т.н., с.н.с
Чемерис Олександр Анатолійович _____

Рецензент:
Посада, науковий ступінь, вчене звання,
Прізвище, ім'я, по батькові _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2021 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра автоматики та управління в технічних системах

Рівень вищої освіти – другий (магістерський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-наукова програма «Інженерія програмного забезпечення комп'ютерних систем»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Олександр, РОЛІК

«___»_____2021 р.

ЗАВДАННЯ

на магістерську дисертацію студенту

Поліщука Мирослава Ігоровича

1. Тема дисертації «Організація програмних систем надання сервісів в музичній галузі», науковий керівник дисертації, доцент кафедри АУТС, к.т.н., затверджені наказом по університету від «12» березня 2021 р. № 809-с

2. Термін подання студентом дисертації 11.05.2021

3. Об'єкт дослідження побудова системи на базі технологій та фреймворків для односторінкових програмних продуктів

4. Предмет дослідження інформаційна система на основі односторінкового застосунку в музичній сфері. Аналіз предметної галузі та адаптування під її потреби односторінкового застосунку

5. Перелік завдань, які потрібно розробити: проаналізувати теоретичні матеріали, визначити та дослідити технології для реалізації, спроектувати модель системи сервісів, спроектувати та реалізувати систему надання послуг згідно розробленої моделі, перевірити за допомогою тестів роботу програми

6. Орієнтовний перелік графічного (ілюстративного) матеріалу: схема архітектури односторінкового застосунку, таблиця швидкості завантаження системи в браузері, деревовидна структура головного контейнера стану, діаграма сценарія виконання користувача в системі,

7. Орієнтовний перелік публікацій Внутрішній цільовий фішинг, Управління пам'яттю в JavaScript

9. Дата видачі завдання 01.02.2021

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Вибір та узгодження теми магістерської дисертації	02.02.2021-04.02.2021	
2	Аналіз теоретичних матеріалів та вивчення предметної області	05.02.2021-18.02.2021	
3	Розробка технічного завдання	15.03.2021-18.03.2021	
4	Проектування моделі системи музичних сервісів	19.03.2021-25.03.2021	
5	Вибір технологій реалізації системи	26.03.2021-28.03.2021	
6	Реалізація, перевірка та налагодження програми	05.04.2021-23.04.2021	
7	Оформлення текстових та графічних матеріалів	15.04.2021-04.05.2021	
8	Передзахист магістерської дисертації	05.05.2021	
9	Доопрацювання пояснювальної записки та підготовка презентації	06.05.2021-11.05.2021	
10	Захист магістерської дисертації	18.05.2021	

Студент Мирослав, ПОЛІЩУК

Науковий керівник

РЕФЕРАТ

Магістерська дисертація на тему «Організація програмних систем надання сервісів в музичній галузі».

Дисертація містить 135 с. тексту, 37 рисунки, 29 таблиця, 21 джерел та 5 додатків. Вона складається з наступних розділів: перелік умовних позначень, скорочень і термінів, вступ, 5 розділів для основної частини, висновки, перелік посилань та 5 додатків.

Актуальність обраної теми полягає в підвищенні ефективності роботи та швидкодії програмного забезпечення за рахунок покращення архітектурних принципів та примінення в предметі дослідження.

Метою магістерської дисертації є покращення швидкодії та ефективності інформаційних систем побудованих на основі односторінкого застосунку.

Об’єкт дослідження є побудова системи на базі технологій та фреймворків для односторінкових програмних продуктів

Предмет дослідження є інформаційна система на основі односторінкового застосунку в музичній сфері. Аналіз предметної галузі та адаптування під її потреби односторінкового застосунку

В ході роботи було створено односторінковий застосунок з використанням мови програмування JavaScript та фреймворку React.

Вихідна базова система має точки розширення для розвитку та може бути інтегрована. У ході отримання результатів розроблено план стартап-проекту.

Ключові слова: SPA, односторінковий застосунок, React, MVVW

ABSTRACT

Master's dissertation on "Organization of software systems for providing services in the music industry."

The dissertation contains 135 pages. text, 37 figures, 29 tables, 21 sources and 5 appendices. It is created from the following sections: a list of conditional knowledge, soon and terms, introduction, 5 sections for the main part, conclusions, link references and 5 appendices.

The relevance of the chosen topic is to increase the efficiency and speed of the software by improving the architectural principles and application in the subject of research.

The aim of the master's dissertation is to improve the speed and efficiency of the one-sided application system.

The object of research is to build a system based on technology and frameworks for one-page software products

The subject of research is an information system based on a one-page application in the field of music. Analysis of the subject area and adaptation to its needs of one-page applicationDuring the work, a one-page application was created using the JavaScript programming language and the React framework.

The source base system has expansion points for development and can be integrated. In the course of obtaining the results, a plan of the startup project was developed.

Keywords: SPA, one-page application, React, MVVW

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

SPA – це веб-застосунок чи веб-сайт, який вміщується на одній сторінці з метою забезпечити користувачу досвід близький до користування настільною програмою.

MVVW – це шаблон проєктування, що застосовується під час проєктування архітектури застосунків

Фреймворк – це готовий до використання комплекс програмних рішень, включаючи дизайн, логіку та базову функціональність системи або підсистеми.

API – набір визначень підпрограм, протоколів взаємодії та засобів для створення програмного забезпечення

JS – динамічна, об'єктно-орієнтована прототипна мова програмування

DOM – специфікація прикладного програмного інтерфейсу для роботи зі структурованими документами (як правило, документами XML)

ВСТУП

Додаток SPA - це буквально одна сторінка, яка постійно взаємодіє з користувачем, динамічно переписуючи поточну сторінку, а не завантажуючи цілі нові сторінки з сервера. Такими прикладами є односторінкові додатки як: Trello, Facebook, Gmail.

Особливості по яким SPA є популярною в тому, що всі елементи необхідні для функціонування системи знаходяться на одній сторінці. Основні елементи заватанжуються один раз і далі тільки оновлюються окремі частини. При необхідності окремі частини динамічно підгружуються після запиту користувача. При цьому оновлення нових модулів відбувається частково що запобігає непотрібних рендерів сторінки та завантаження великого об'єму даних.

Завдяки цьому збільшується швидкодія програмного забезпечення і скорочує кількість обміну запитів між сервером і клієнтом. Таким чином і серверна частина не отримує велике навантаження.

Метою магістерської дисертації є покращення швидкодії та ефективності інформаційних систем побудованих на основі односторінкого застосунку.

Об'єкт дослідження є побудова системи на базі технологій та фреймворків для односторінкових програмних продуктів

Предмет дослідження є інформаційна система на основі односторінкового застосунку в музичній сфері. Аналіз предметної галузі та адаптування під її потреби односторінкового застосунку

В ході роботи досліджено принципи побудови для односторінкових застосунків. Виділено основні архітектурні шаблони та принципи, ці висновки було здійснені в результаті дослідження. На основі наукових досліджень які були опубліковані досягненно мету дисертації.

Завдяки обраній архітектурі було створенно універсальне рішення для побудови програмного забезпечення.

Так у рамках наукової десертації було продемостровано у ілюстративних матеріалах комплексне рішення побудова системи надання сервісів. Представлене універсальне рішення яке можна використати не тільки у музичній галузі.

Для реалізації системи було обрано музичну предметну область, так як в даній області на даний момент не має аналогу і вона продемонструє всі переваги даної архітектури на якій побудована система. А саме швидкодію системи та односторонній потік даних. Це зробить інтерфейс користувача дуже швидким навіть при повільному інтернеті. Здійсненно аналіз можливих або схожих додатків для створення власного графічного інтерфейсу.

Таким чинном, наукову новизну роботи забезпечує такі чинники: вибір певних шаблонів для проектування та їх комбінації, дослідження управління пам'ят для досягнення швидкодії, проектування унікального графічного інтерфейсу

Конкретні завдання роботи:

- Дослідження створення односторінкових застосунків. Аналіз існуючих архітектурних рішень.;
- огляд та аналіз предметної області;
- проектування моделі сервісів
- розроблення сервісів в односторінковому застосунку відповідно на проектної моделі
- створення програмно реалізації ;
- тестування отриманих результатів та виконаною цілі мети.

Публікації. Результати проведених у рамках магістерської дисертації досліджень було оприлюднено на IV МІЖНАРОДНОЇ НАУКОВО-ПРАКТИЧНОЇ КОНФЕРЕНЦІЇ “ПРОБЛЕМИ КІБЕРБЕЗПЕКИ ІНФОРМАЦІЙНО-ТЕЛЕКОМУНІКАЦІЙНИХ СИСТЕМ” (PCSITS), а також в Наукова Конференція Європейська наука XXI століття

1 ПРОЕКТУВАННЯ ПРОГРАМНОЇ СИСТЕМИ

1.1 Односторінковий застосунок

Односторінковий застосунок – веб-застосунок або веб-сайт, в якому вміщується но одній сторінці з метою приближення користування до настільною програмою.

Односторінкові застосунки створенні для приведення потужності настільного додатка до міжплатформеного середовища веб-браузера. Односторінкова (веб-програма), або SPA, розроблена саме для цього. На створення SPA впливало AJAX та стрімка його популярність.

AJAX- підхід до побудови користувацьких інтерфейсів веб-застосунків, за яких вебсторінка, не перезавантажуючись, у фоновому режимі надсилає запити на сервер і сама звідти довантажує потрібні користувачу дані. AJAX — один з компонентів концепції. Використовуючи даний підхід для динамічної маніпуляції з обновленням об'єктної моделі документа (DOM) та використання CSS для зміни стилю сторінки на льоту, висунуло AJAX на перший план шаблони та практики зазвичай використовується при створенні SPA може призвести до загальної ефективності в розробці додатків, обслуговування коду та час розробки

Як і у більшості нових рішень, односторінковий дизайн додатків включає різні підходи. Різні думки сучасних експертів, а також безліч конкуруючих бібліотек та фреймворків, можуть знайти правильне рішення для вашого SPA-проекту .

1.2 Архітектура SPA

У SPA весь додаток працює як одна веб-сторінка. У цьому підході рівень представлення для всієї програми був вилучений із сервера і є керується з браузера. Завдяки такому підходу кожен запит на новий вигляд (HTML-сторінка)

призводить до зворотнього запиту на сервер. Коли потрібні свіжі дані на стороні клієнта, запит надсилається до сторона сервера. На стороні сервера запит перехоплюється об'єктом контролера всередині презентаційного шару. Потім контролер взаємодіє з шаром моделі через сервісний рівень, який визначає компоненти, необхідні для завершення шару моделі завдання. Після отримання даних або об’єктом доступу до даних (DAO), або службою агента, будь-які необхідні зміни до даних вноситься бізнес-логікою в діловий рівень.

Управління передається назад на рівень презентації, де вибирається відповідний вигляд. Логіка презентації диктує, як свіжоотримані дані ввідображаються в вибраний вигляд. На рисунку 1.1зображено вся логіка яку виконує додаток[3].

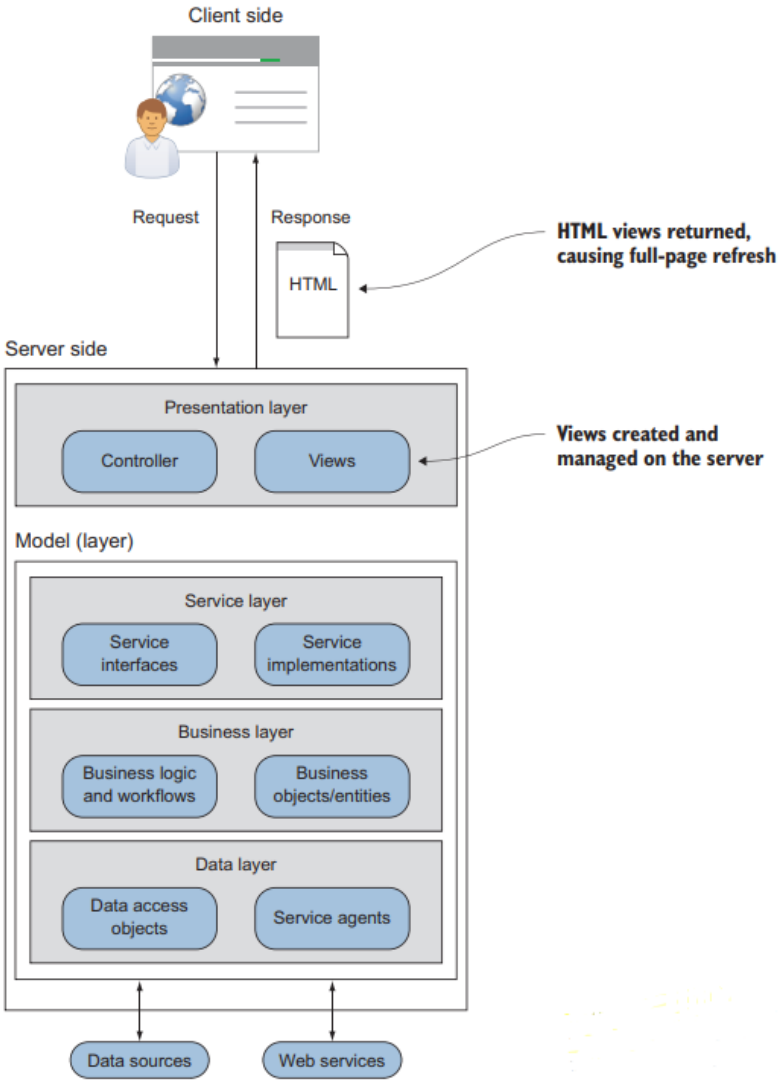


Рисунок 1.1 — Вигляд SPA

Часто отримане подання(відображення) починається як вихідний файл із заповнювачами, де потрібно вставити дані (і, можливо, інші інструкції щодо візуалізації). Цей файл діє як різновид шаблону для того, як подання відображається, коли контролер направляє на нього запит. Після об'єднання даних та відправлення подання повертається до браузера. Потім браузер отримує нову сторінку HTML, і за допомогою оновлення інтерфейсу користувач бачить нове подання, що містить запитувані дані. Весь процес зображено на рисунку 1.2[3]

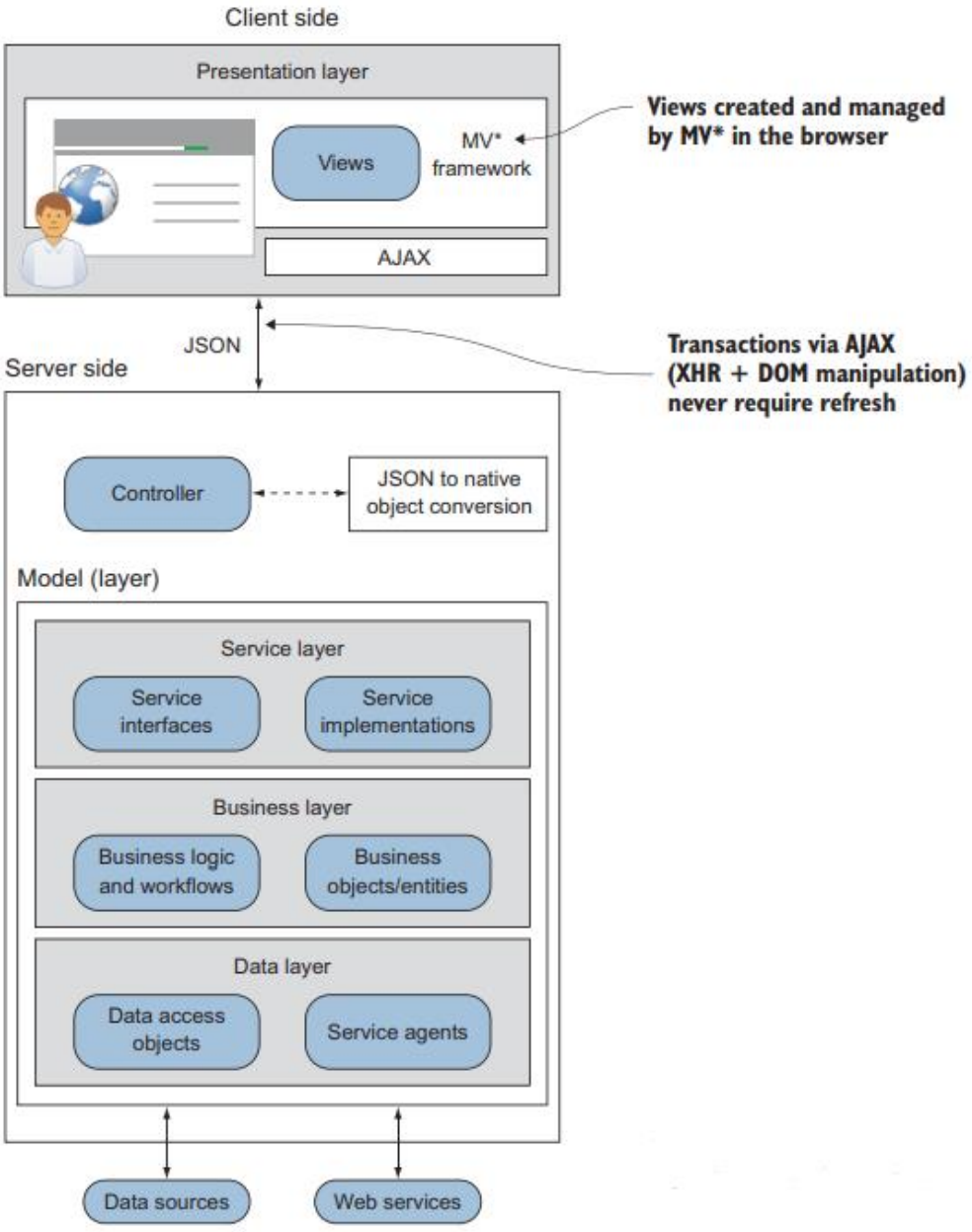


Рисунок 1.2 — Отримання даних та оновлення сторінки

Це демонструє, як цей дизайн може виглядати як SPA. Зверніть увагу на те, що сталося з рівнем презентації та нашими транзакціями. Переміщення процесу створення та управління поданнями з одного інтерфейсу через інший інтерфейс роз'єднує його з процесом сервера. З архітектурної точки зору це дає SPA цікаві переваги. Якщо ви не робите частковий візуалізацію на сервері, серверу більше не потрібно брати участь у способі подання даних.

Загальний дизайн SPA майже такий самий, як і традиційний дизайн. Ключові зміни полягають у наступному: повного оновлення браузера не відбувається, логіка презентації знаходиться в транзакції клієнта, а серверні транзанції можуть бути лише даними, залежно від ваших уподобань візуалізація даних

1.3 Логіка відображення в клієнті та оновлення браузера

У SPA перегляди не є повними HTML-сторінками. Вони є лише частинами DOM які складають видимі області екрана. Після початкового завантаження сторінки всі інструменти необхідні для створення та відображення подань одни раз завантажуються та готові до використання. Якщо нове представлення потрібне, він генерується локально у браузері та динамічно за допомогою API приєднується до DOM через JavaScript. Ніколи не потрібні повні оновлення браузера. [1]

Оскільки наша логіка презентацій - це переважно сторона клієнта в SPA, завдання комбінування HTML і дані переміщуються з сервера в браузер. Як і на стороні сервера, джерело HTML містить заповнювачі, куди слід вставити дані (і, можливо, інші інструкції щодо відтворення)[4]. Цей шаблон на стороні клієнта використовується як основа для видалення нового перегляди у клієнта. Однак це не шаблон HTML для цілої сторінки. Це лише для частина сторінки, яку представляє подання.

Важкий підйом маршрутизації до правильного подання, поєднуючи дані з HTML шаблон, а управління життєвим циклом подання, як правило, делегується сторонньому файлу JavaScript, який зазвичай називають структурою MV * .

1.4 Модульність SPA

Для інкапсуляції програмних компонентів використовується модулі, за допомогою них досягається функціональність програмного продукту в модулях.

Модулі - це спосіб групувати різні частини функціональності, приховуючи одні частини, викриваючи інші. У ECMAScript 6 і вищих версіях JavaScript, модулі підтримуватися вбудовано. У традиційному додатку, коли сторінка перезавантажується.Усі попередні створені об'єкти JavaScript стираються, і створюються об'єкти для нової сторінки. Це не тільки звільняє пам’ять для нової сторінки, але також гарантує, що імена функцій та змінних сторінки не мають жодних шансів конфлікту з іншими сторінками. [18]

Це не стосується односторінкового застосунку. Наявність однієї сторінки означає, що ви не створюєте весь цей процес заново, коли користувач запитує новий перегляд. Модулі допоможуть вам виправити цю проблему. Модуль обмежує область вашого коду. Змінні та функції, визначені всередині кожен модуль має область, яка відповідає локальній структурі. На рисунку 1.3 зображено переваги модульної системи. [5]

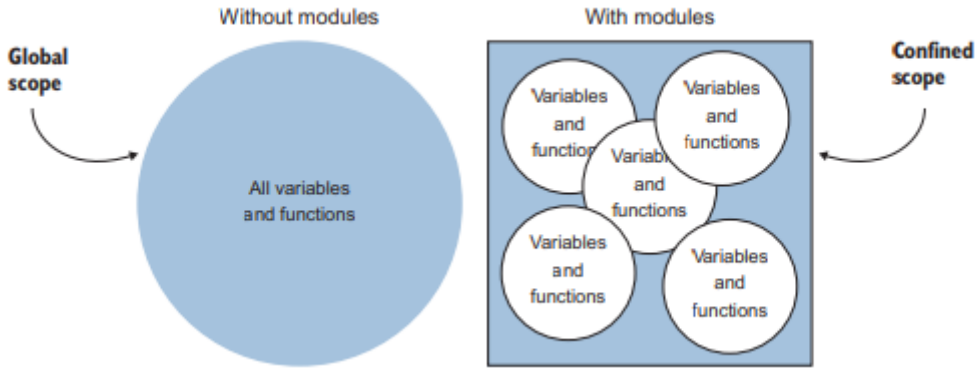


Рисунок 1.3 — Порівняння модульної системи

Шаблон модуля в зв’зку з іншими методами управління надає спосіб розробляти великі, надійні веб-програми з односторінковою архітектурою.

Модулі інкапсулюють нашу логіку та. Хоча це допомагає розв’язати і приватизувати код програми, все ще потрібен спосіб для комунікації модулів між собою.

1.5 Композиція та макет

У односторінковій програмі інтерфейс користувача будується з поданнями замість нових сторінок. Створення регіонів вмісту та розміщення переглядів у цих регіонах визначити макет вашої програми. Клієнтська маршрутизація використовується для з'єднання точок. Всі ці елементи поєднуються, щоб вплинути як на зручність використання програми, так і на її естетична привабливість.

1.6 Комунікація з серверною частиною

Веб-сторінки зазнали сильних змін з часу введення API XMLHttpRequest. В основі лежить колекція методів AJAX, які обертаються навколо цього API. Можливість асинхронного отримання даних та перефарбовування частин екрана є основним елементом односторінкової архітектури. Зрештою, в SPA ми створюємо певного роду обманку для користувачів, які в процесі навігації екран змінюється плавно і невимушено. Здійснюється це завдяки навігації в SPA, а саме за допомогою History API.

Тож яким би був цей подвиг демонстрації програми, не маючи можливості отримувати дані для наших користувачів.

1.7 Навігація в SPA

DOM SPA зазвичай починається як оболонка в індексі SPA сторінки. Це все, що потрібно. Модулі SPA та фреймворк MV *, включаючи бібліотеки, що підтримують, завантажуються разом із індексною сторінкою або асинхронно завантажується, якщо використовується бібліотека для завантаження сценарію AMD. SPA також має можливість асинхронно отримувати дані та будь-які віддалені шаблони (часткові) та інші ресурси, які ще не були включені та для динамічної візуалізації подань за потреби.

Під час навігації користувач безперервно відображає подання. Це в поєднанні з асинхронним вибором даних надає додатку плавну навігацію переходу візуальна частина роботи відображена на рисунку 1.4, схоже на традиційну для забезпечення чудового користувацького досвіду. [3]

Рисунок 1.4 — Маршрутизатор в SPA

Однак після завантаження SPA користувачі потребують способу доступу до додаткового вмісту за допомогою переходів в межах одного програмного продукту. Оскільки SPA все ще є веб-додатком, користувачі будуть очікувати, що зможуть використовувати для навігації адресний рядок та навігаційні кнопки браузера.

Це здійснюється за допомогою компонент JavaScript, який робить навігацію в цьому односторінковому середовищі можливим називається клієнтським маршрутизатором (або просто маршрутизатор).

Маршрутизатор управляє станом програми, беручи на себе контроль навігації браузера, дозволяючи розробникам безпосередньо зіставляти зміни в URL-адресі з функціональністю на стороні клієнта. Використовуючи цей підхід, не потрібні зворотні запити до сервера.

Маршрутизатор визначає, коли зміна стану необхідна за допомогою різних методів виявлення змін у місцезнаходження браузера, наприклад прослуховування певних подій. Будь-коли зміна URL-адреси трапляється, маршрутизатор намагається зіставити частину нової URL-адреси із записом у її конфігурації.

Перш ніж розбивати типові частини конфігурації маршрутизатора. На рисунку 1.5 наведено огляд процесу навігації в SPA та підкреслює роль маршрутизатора на стороні клієнта[5]. Зверніть увагу, що маршрутизатор жодного разу не працює через взаємодію з сервером. Вся маршрутизація здійснюється у браузері.

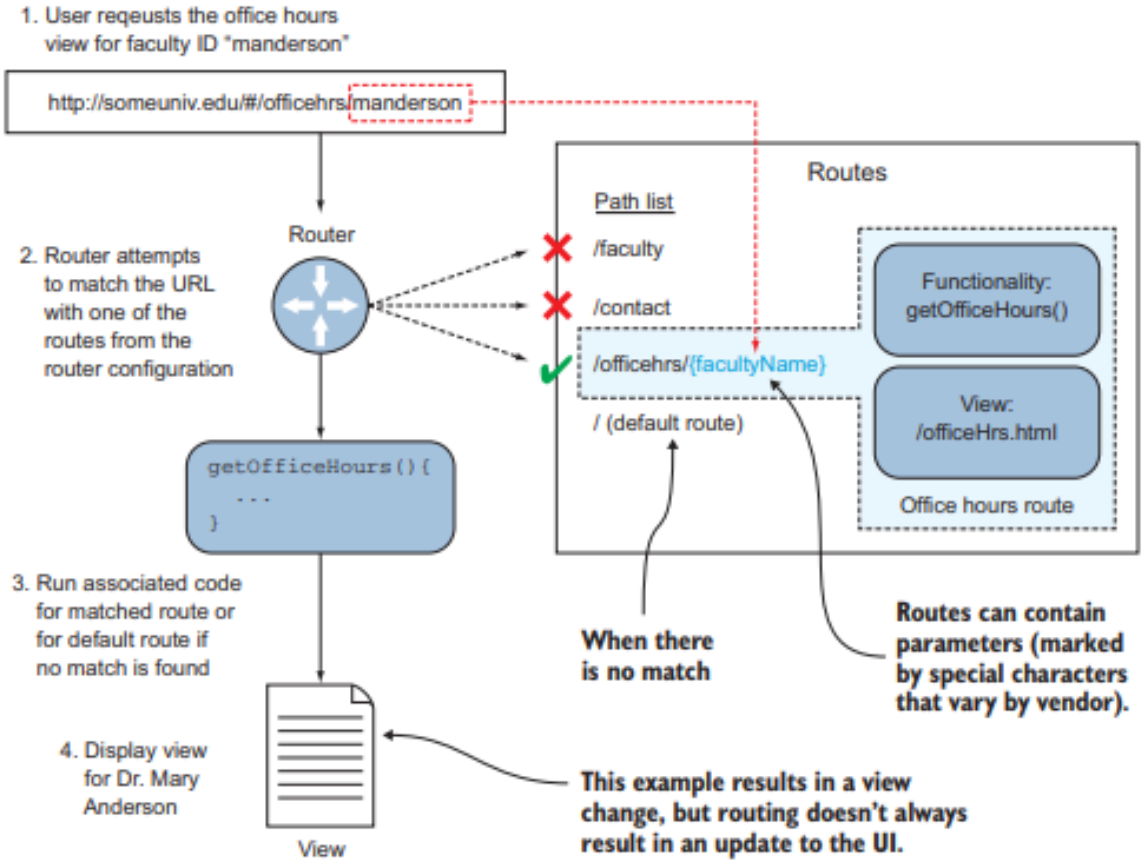


Рисунок 1.5 — Навігаційна система SPA

На рисунку 1.5 зображено вся система навігації , коли маршрутизатор збігається із конфігурацією шляхів із реальними URL-адресами у браузері, він може визначити, які типи змін у стані програми повинні відбуватися.

1.8 Аналіз існуючих інструментів

З метою протидії складності та сприяння розвитку SPA, протягом багатьох років було створено важливий інструмент, який називається SPA фреймворк. Він надає розробнику структуру програмування, що складається з бібліотек, включаючи їх функціональні можливості та методи для підтримки їх у впровадженні SPA. Завдяки цьому SPA фреймворки постійно оновлюються та вдосконалюються. Через ці зміни на сьогодні стало досить важко зрозуміти. Для кращого розуміння будуть представлені та оцінені три найпопулярніші рамки SPA на основі заздалегідь визначених вимог[1].

Angular використовує архітектуру на основі компонентів, яка забезпечує високий ступінь багаторазового використання компонентів протягом усього додатку і характеризується як високою якістю коду, так і високою продуктивністю, що обумовлено такими факторами, як введення ієрархічної залежності або Angular Universal. Angular також використовує попередній візуалізацію на стороні сервера, що гарантує, що всі пошукові системи отримують доступ до вмісту програми, а додатки соціальних мереж можуть правильно відображати попередній перегляд сайту.

Таким чином, Angular ідеально розроблений для додатків на рівні підприємства. Це пов’язано як із TypeScript, інструментарій якого забезпечує розширені послуги автоматичного заповнення, навігації та рефакторингу, так і з багатослівністю середовища SPA, що передбачає велику криву навчання та великі затрати часу.

Плюси:

- Незалежність платформи
- Архітектура на основі компонентів
- Машинопис
- Довгострокова підтримка Google

React

На відміну від Angular, який забезпечує ефективну структуру SPA, React - це бібліотека з відкритим кодом JavaScript, що включає компілятор JSX, який в

основному орієнтований на користувальницький інтерфейс і дозволяє створювати компоненти, що використовуються багаторазово. Таким чином, React пропонує рішення, орієнтоване на користувальницький інтерфейс, яке в кінцевому підсумку пропонує користувачам дуже чуйний та гнучкий інтерфейс завантаження.

Рішення React є зручними для SEO та одночасно пропонують високу продуктивність та гнучкість завдяки використанню віртуального DOM. Повторне використання компонентів та односпрямований потік даних, що гарантує стабільність програмного коду, також робить програмування одночасно точним та зручним для розробника.

Однак, оскільки React - це просто бібліотека JavaScript, слід зазначити, що він більше підходить для розробки менших додатків, оскільки може бути важко вибрати та зібрати правильні інструменти для великих програм.

Плюси:

- Віртуальний DOM
- Багаторазове використання компонентів
- Гнучкість
- Односпрямований потік даних
- Підтримка громади

Vue

Наприклад, Vue пропонує двонаправлене прив'язку даних, рендеринг на стороні сервера, підтримку TypeScript та JSX та подібну до Angular архітектуру на основі компонентів. Всі ці функції поєднані в зручний, легкий пакет, який дозволяє Vue перевершити React або Angular як за розміром, так і за продуктивністю.

Крім того, Vue вражає своєю гнучкістю в потоці даних, оскільки підтримує як двонаправлений зв'язок, що робить обробку HTML-блоків простішим і швидшим, так і односпрямований зв'язок, що важливо при роботі з різними компонентами. Як і React, Vue також використовує гнучку екосистему, яку можна налаштувати відповідно до вимог програми будь-якого розміру, що надає

розробнику велику свободу та можливість поєднувати Vue з різними бібліотеками.

Однак занадто велика свобода та гнучкість також можуть мати недолік у процесі прийняття рішень, оскільки розробник повинен знайти найбільш підходящу бібліотеку серед безлічі корисних бібліотек.

Загалом, Vue особливо підходить для швидкого розвитку кроссплатформенних рішень. Він забезпечує основу для складних SPA-центрів, а також корисне рішення для додатків, де продуктивність хорошої кодової організації або структури додатків є першорядними.

Плюси:

- Архітектура на основі компонентів
- Гнучкість
- Простота
- Маленький розмір

Аналіз

1.9 Вибір фреймворку

Багато інтерфейсних програм покладаються на глобальне управління державою для зберігання інформації, наприклад, хто ввійшов у систему та інші налаштування користувача. Найпопулярнішим проектом управління станом JavaScript є Redux. Більшість розробників використовують офіційні прив'язки React для Redux, які підтримуються командою Redux.

Вибираючи фреймворк або бібліотеку, потрібно думати і про продуктивність. У багатьох випадках вам не доведеться турбуватися про ефективність, особливо якщо ви будуєте невеликий проект. Однак, чим більше проект розширюється за масштабами та складністю, продуктивність може (і буде) викликати занепокоєння. Дані про швидкодію фреймворку представлено на

рисунку 1.6 і на ньому видно, що React має найбільшу продуктивність з усіх інших.

Name Duration for...	vue-v2.6.2-keyed	angular-v8.2.14-keyed	react-v16.8.6-keyed
create rows	160.9 ± 2.4	157.3 ± 3.0	176.2 ± 3.9
replace all rows	133.1 ± 1.2	137.5 ± 1.4	145.7 ± 1.3
partial update	223.8 ± 7.9	152.3 ± 3.4	218.7 ± 2.8
select row	305.9 ± 10.7	74.8 ± 2.7	124.6 ± 5.1
swap rows	69.9 ± 2.8	441.7 ± 2.4	445.5 ± 2.1
remove row	29.0 ± 0.5	27.7 ± 0.8	27.6 ± 2.1
create many rows	1,300.6 ± 15.1	1,406.4 ± 36.5	1,827.0 ± 75.1
append rows to large table	310.2 ± 4.5	303.3 ± 13.0	345.9 ± 6.8
clear rows	156.3 ± 1.1	248.6 ± 3.3	149.2 ± 2.1
slowdown geometric mean	1.24	1.32	1.46

Рисунок 1.6 — Результати тесту

Perf Track від Google Chrome Labs показує виробничі дані з тисяч веб-сайтів. На цю статистику впливає багато іншого, і не лише рамки вибору.

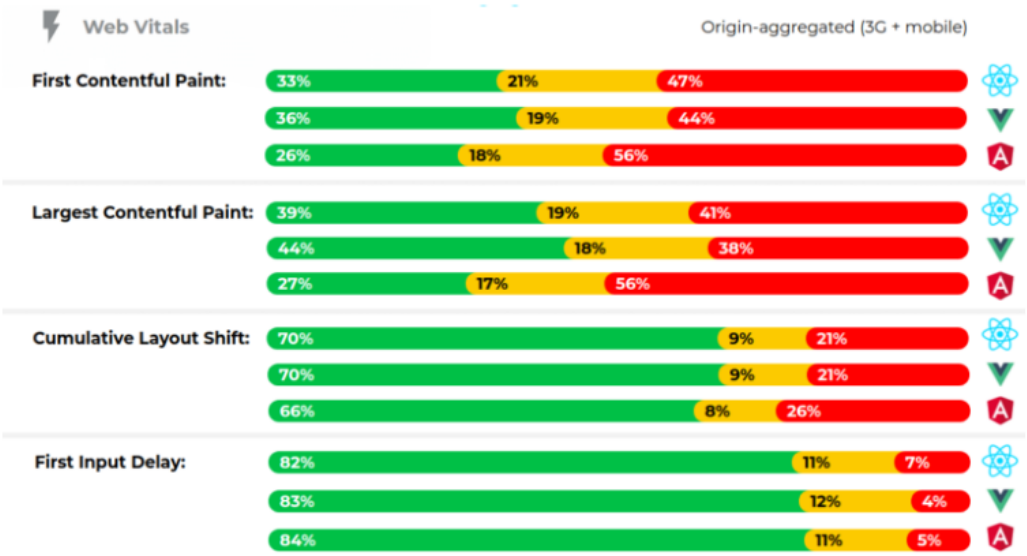


Рисунок 1.7— Результати Perf Track

First Contentful Paint веб-сайти Vue та React вищі за цією метрикою порівняно з Angular, що може зайняти більше часу для завантаження та представлення вмісту користувачеві.

Largest Contentful Paint Angular також є найповільнішим серед трьох фреймворків при отриманні повної сторінки, лише 27 відсотків веб-сайтів Angular бали в прийнятному діапазоні.

First Input Delay для всіх трьох фреймворків понад 80 відсотків веб-сайтів знаходяться в допустимому діапазоні для першої затримки введення, що показує, скільки потрібно, поки користувач не зможе взаємодіяти зі сторінкою.

JavaScript Bytes найбільш легкими є програми, розроблені з Vue, 68 відсотків програм Vue завантажують менше 1 Мб JavaScript. З іншого боку, програми Angular та React, як правило, мають більший розмір коду. [17]

Завдяки проведеному дослідженню було обрано для реалізації фреймворк React через ряд причин перечислених вище. Для отримання максимальної швидкодії та більшої свободи у прийнятті рішення.

1.9.1 React

React - це JavaScript-бібліотека для створення користувацьких інтерфейсів. Virtual DOM, або віртуальний DOM, - легка копія реального HTML DOM у вигляді JS-об'єкта в пам'яті, з якої працює додаток. Він агрегує в собі всі динамічні зміни, а вже після застосовує їх до реального DOM.

Основна проблема нативного HTML DOM - його швидкість при роботі з маніпуляцією динамічними даними: з нього дорого як читати, так і багато і часто писати. Віртуальний DOM не є простим об'єктом, який зберігає дані, він має ефективні і продуктивні алгоритми для порівняння і угрупування змін і вибіркових змін окремих частин HTML DOM.

Використання SPA є створення веб-програми яка виглядає як нативна програма. Використовуючи методи застосування SPA, потрібно змусити

програму відповісти і виглядати так, ніби вона була встановлена на пристрої. Тому, наскільки як може зрозуміти користувач, програма SPA є власною програмою для пристроїв[16]. Є певні особливості та вимоги, які завжди будуть загальними для всіх:

- Як випливає з назви, вся програма живе лише на одній HTML-сторінці. На відміну від стандартних, Додатки HTML, які використовують окремі сторінки для відображення різних екранів, перша сторінка – це єдина сторінка, яка коли-небудь завантажується у програму SPA
- Замість статичних HTML-файлів JavaScript динамічно відображає один файл. Отже, HTML-сторінка, яка завантажується вперше, насправді майже повністю позбавлена вмісту. Але те, що він матиме, - це кореневий елемент всередині тегу body, який стає контейнер для всієї програми, яка знову відображається в режимі реальної дії під час взаємодії користувача з додатком.
- Усі сценарії та файли, необхідні для запуску програми, зазвичай завантажуються в початку, при отриманні основного файлу HTML. Однак цей метод змінюється та інші програми завантажують лише файл сценарію базового рівня, а потім завантаження інших сценаріїв на вимогу за потребою.
- Маршрутизація URL-адрес обробляється для SPA. У програмах SPA є деякі механізм, який використовується, залежно від обраної вами структури, для створення віртуальна маршрутизація. Віртуальна маршрутизація просто означає, що хоча вона і здається користувач, який робить різні запити на різні URL-адреси на стороні сервера, насправді, вся маршрутизація відбувається лише в клієнтському браузері для того, щоб зробити логічним переходи на різні екрани. Іншими словами, ніяких запитів на сервери не робиться і Маршрутизація URL стає засобом логічного розділення програми на різні екрани. Наприклад, коли користувач вводить URL-адресу у свій браузер, він повинен натиснути Введіть для того, щоб подання було відправлено назад на сервер, для якого призначено URL. Однак у випадку маршрутизації, що відбувається в програмі SPA, фактичного фактичного немає шлях до сервера,

вказаний URL-адресою. Він не існує. Тому зворотного звороту ніколи не буває спрацьовує. Натомість програма використовує URL-адресу як свого роду контейнер для розділів програми, а також викликати певну поведінку, коли є певні URL-адреси дано. Сказавши це, маршрутизація URL-адреси все ще є корисною функцією, оскільки маршрутизація є очікувані можливості більшості користувачів, і це дозволяє їм робити закладки екранів.

Інший новий концепт, який привніс React, -JSX (Javascript and XML). Це синтаксичний змінений JavaScript або, по-іншому, розширення JavaScript. JSX нагадує мову шаблонів.

React реалізує архітектурні патерни MVC і MVVM, так ось React і його компоненти - це буква "V" в MVC / MVVM, тобто view або уявлення. Такий підхід дозволяє писати компоненти які можна повторно використовувати і збирати великі функціональні блоки за допомогою композиції більш дрібних компонентів. [14]

React має дві основні конструкції:

- React підтримує власний віртуальний DOM під час виконання. Цей віртуальний DOM відрізняється від DOM браузера. Це унікальна унікальна копія DOM, яку він створює та підтримує на основі інструкцій з нашого коду. Цей віртуальний DOM створений та редагується за потребою на основі процесу узгодження, який робить служба React внутрішньо.Цей процес узгодження - це процес порівняння, на який дивиться React браузер DOM і протиставляє цьому свій власний віртуальний DOM. Це процес зазвичай відомий як фаза візуалізації. Коли виявляються відмінності— наприклад, віртуальна DOM містить деякий елемент, який не міститься у браузері DOM - React надішле певні інструкції браузеру DOM, щоб створити цей елемент так що DOM браузера та віртуальний DOM збігаються. Цей процес додавання, редагування, або видалення елементів зазвичай називається як фаза фіксації.
- У React, додаток складається з багатьох компонентів, і в кожному компоненті вони можуть бути якась внутрішній стан (state). Якщо ці дані змінюються з

будь-якої причини, React спрацьовує процес рендеру компонента та за потреби внесіть зміни в DOM. Щоб внести нові дані до обраного компонента.

Можна описати форму, яка складається з різних атомарних компонентів на JSX: За допомогою такого компонентного підходу і композиції можна формувати уявлення будь-якої складності.

React надає хороший огляд механізму звірки на високому рівні: роль елементів React, методи життєвого циклу і метод render, а також диференціює алгоритм, який застосовується до дочірнім компонентів. Дерево іммутабельних елементів React, що повертається з методу render, зазвичай називається "віртуальним DOM" (virtual DOM). Цей термін допоміг пояснити React людям на ранній стадії, але він також викликав нерозуміння і більше не використовується в документації React. [15]

Крім дерева елементів React, у фреймворку завжди було дерево внутрішніх елементів (компоненти, DOM Ноди і т.д.), які використовуються для зберігання Стейту. Починаючи з версії 16, React випустив нову реалізацію цього дерева внутрішніх елементів і алгоритму під назвою Fiber, який керує ним.

1.9.2 Fiber

При наведенні даних від кожного елемента React, повернення методом візуалізації, сливаються в дерево fiber нод. Кожен елемент React має відповідну fiber ноду. Відмінність від элементів React, fiber ноди не створюються заново при кожному рендерингу. Це мутабельні структури даних, що містять стан компонентів та DOM.

В залежності від типу React елемента фреймворк повинен виконувати різні типи завдань. Він визначає методи життєвого цикла та метод рендерингу, в той час як для хост-компонентів (DOM ноди) він виконує мутацію DOM. Таким чином, кожен елемент React перетворюється у fiber ноду, що відповідає типу, який описує завдання, які необхідно виконати. [6]

Можна думати про fiber ноді як про структуру даних, яка представляє собою деяку роботу, або, іншими словами, одиницю роботи. Архітектура fiber також забезпечує зручний спосіб відстеження, планування, припинення і переривання роботи.

Коли React елемент вперше перетвориться в fiber ноду, React використовує дані цього елемента для створення fiber в функції (createFiberFromTypeAndProps). В оновленні React повторно використовує fiber ноди і просто оновлює необхідні властивості, використовуючи такі дані від конкретного елемента React. Також може знадобитися перемістити ноду в ієрархії на основі пропса key або видалити його, якщо відповідний елемент React більше не повертається з методу render.

Таким чином React створює fiber ноду для кожного React елемента і оскільки утворюється дерево цих елементів, то буде дерево fiber нод. [6]

Всі fiber ноди з'єднуються через пов'язаний список, використовуючи такі властивості: дочірні, child, sibling і return

Current і WorkInProgress дерева fiber нод. Після першого рендеринга React отримує дане дерево fiber нод, яке відображає стан додатку, що використовувався для рендеринга. Це дерево елементів часто називають current. Коли React починає працювати над оновленнями, він будує так зване дерево WorkInProgress, яке відображає певний майбутній стан, яке має в швидкому майбутньому виводитися на екран. [7]

Всі відповідні роботи виконуються на нодах з дерева WorkInProgress. Коли React проходить через current дерево, для кожної існуючої Ноди він створює подібну альтернативну, яка становить дерево WorkInProgress. Ця нода створюється з використанням даних з React елементів, що повертаються render методом. Як тільки оновлення будуть оброблені і всі пов'язані з ними роботи будуть завершені, React матиме окреме альтернативне дерево, готове до відображення стану на екрані. Як тільки дерево WorkInProgress відображається на екрані, воно стає current деревом або головним деревом.

Одним з ключових принципів React є впорятдкована послідовність. React завжди оновлює DOM за один прохід - він не показує ніяких часткових результатів. Дерево WorkInProgress служить для користувача "чернеткою", так що спочатку React може обробляти всі компоненти, а потім показувати їх зміни на екрані. [7]

У вихідному коді ви побачите безліч функцій, які використовують fiber Ноди як з current дерева, так і з дерева WorkInProgress. Ось одна з таких функцій.

Кожна fiber нода містить посилання на свій аналог з іншого дерева в полі alternate. Нода з current дерева вказує на ноду з дерева WorkInProgress і навпаки.

React компонент можна розглядати як функцію, яка використовує якийсь стейт і пропси для обчислення уявлення. Будь-яку іншу дію, таке як мутація DOM або виклик методів життєвого циклу, слід розглядати як побічний ефект або, просто кажучи, ефект.

Завантаження даних, підписку або зміни в дереві DOM-елементів вручну з React. Такі маніпуляції та операції ще називають "побічними ефектами" (або "ефектами"), тому що вони можуть вплинути на інші компоненти незалежні і не можуть бути виконані під час рендеринга.

Ви можете побачити, як більшість оновлень стейт і пропсов призводять до побічних ефектів. А оскільки застосування ефектів - це різновид роботи, fiber нода є зручним механізмом для відстеження побічних ефектів на додаток до оновлень. Кожна fiber нода може мати ефекти, пов'язані з ним. Вони містяться в поле effectTag.

Таким чином, ефекти в Fiber в основному визначають окрему роботу, яка повинна бути виконана для інстанса після його обробки та оновлень. Для хост-компонентів (DOM) робота складається з додавання, оновлення або видалення елементів. Для класових компонентів може знадобитися оновлення силок і виклик методів життєвого циклу componentDidMount і componentDidUpdate. Існують і інші ефекти, відповідні іншим типам fiber нод.

React реагує на поновлення процесів дуже швидко і для досягнення такого рівня продуктивності використовує кілька цікавих методик. Одним з них є побудова зв'язного списку fiber нод з ефектами для швидкої ітерації. Ітерація зв'язного списку відбувається набагато швидше, ніж дерево, і немає необхідності витрачати час на Ноди без побічних ефектів. [8]

Метою цього списку є позначати Ноди, які мають поновлення DOM або інші ефекти, пов'язані з ними. Цей список є підмножиною дерева finishedWork і пов'язаний з ними допомогою властивості nextEfect замість властивості child, використовуваного в current дереві і дереві WorkInProgress. [8]

Щоб візуалізувати це, на рисунку собі наступне дерево fiber нод, де виділені на рис. 1.8 вузли мають деяку роботу. Наприклад, оновлення призвело до того, що c2 був вставлений в DOM, d2 і c1 повинні змінити атрибути, а b2 потрібно запустити метод життєвого циклу. Список ефектів зв'яже їх разом, щоб React міг пропустити інші Ноди пізніше.

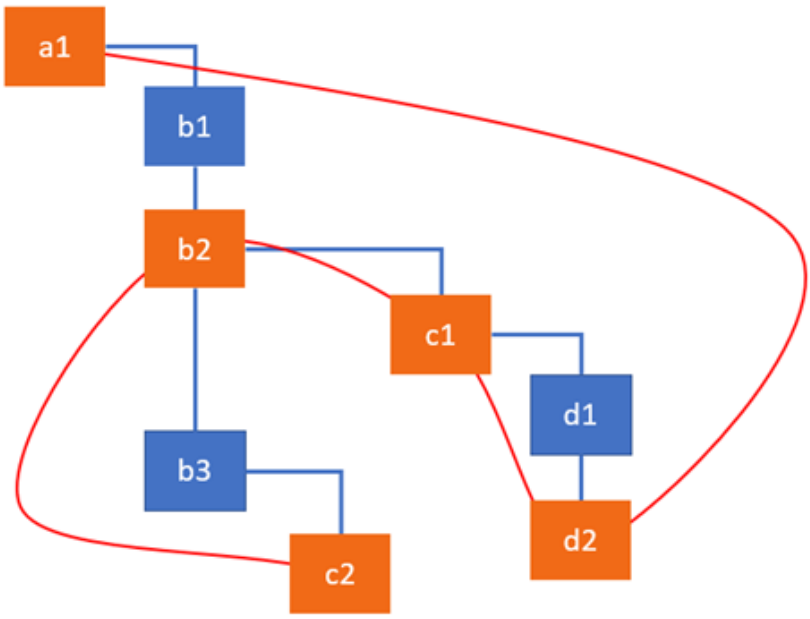


Рисунок 1.8 Дерево fiber нод

На рис.1.8 Дерево fiver нод видно що ноди з ефектами пов'язані між собою. При переборі нод React використовує покажчик firstEffect, щоб визначити, звідки

починається список. Таким чином, діаграма вище може бути представлена увигляді зв'язного списку .

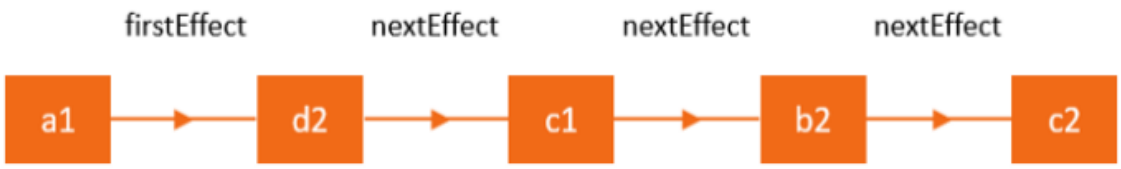


Рисунок 1.9 Діаграма зв'язного списку

Як зображено на рис. 1.9 діаграма зв'язного списку, React діє в порядку від дітей і до батьків. [8]

Коренева нода fiber дерева. Кожен React додаток містить один або кілька DOM елементів, які діють як контейнери. Div елемент з ID container. React створює кореневу fiber ноду для кожного з цих контейнерів. Отримати до неї доступ, використовуючи посилання на DOM елемент. Ця коренева fiber нода зберігає посилання на fiber дерево. Вона зберігається в current властивості fiber Ноди. Дерево fiber нод починається зі спеціального типу Ноди, яким є HostRoot. Він створений всередині і виступає в якості батьківського компонента для вашого самого верхнього компонента. Є зв'язок між вузлом HostRoot і FiberRoot через властивість StateNode. [7]

Fiber дерево, отримавши доступ до самої верхньої ноді HostRoot через кореневу fiber ноду. Або ви можете отримати окрему fiber ноду з такого інстанси компонента, як цей.

Структура fiber Ноди - у fiber нод досить багато полів. Призначення полів alternate, effectTag inextEffect. Тепер давайте подивимося, навіщо нам потрібні інші.

stateNode- містить посилання на інстанси класу компонента, DOM Ноди або іншого типу елемента React, пов'язаного з fiber нодою. Загалом, можна сказати, що ця властивість використовується для зберігання локального стану, пов'язаного з fiber нодою.

Type- визначає функцію або клас, пов'язаний з цією fiber нодою. Для класових компонентів він вказує на конструктор, а для DOM елементів вказує на певний HTML-тег.

Визначає тип fiber Ноди. Використовується в алгоритмі звірки для визначення того, які роботи необхідно виконати. Як згадувалося раніше, робота залежить від типу компонента або ноди в залежності від типу React елемента. Функція createFiberFromTypeAndProps зіставляє React елемент з відповідним типом fiber Ноди. А саме властивість tag для ClickCountercomponent дорівнює 1, що позначає ClassComponent, а для span-елемента - 5, що означає HostComponent. updateQueue. [6] Черга оновлень Стейт, коллбеків і оновлень DOM. memoizedState Стан fiber Ноди, яке використовувалося для виведення на екран. При обробці оновлень відображається поточний стан, що відображається на екрані. memoizedProps Пропси fiber Ноди, які були використані для виведення під час попереднього рендеру. pendingProps Пропси, які були оновлені на основі нових даних в React елементах і повинні бути застосовані до дочірнім компонентів або DOM елементів. key

Унікальний ідентифікатор групи дочірніх елементів, щоб допомогти React з'ясувати, які елементи були змінені, додані або вилучені зі списку. Це пов'язано з описаної тут функціональністю "списків і ключів" React.

Під час першого рендеру React застосовує поновлення до компонентів, запланованим через setState або React.render і з'ясовує, що необхідно оновити в інтерфейсі. Якщо це перший рендер, React створює нову fiber ноду для кожного елемента, що повертається з рендера методу[8]. У наступних оновленнях fiber Ноди для існуючих елементів React використовуються повторно і оновлюються. Результатом фази є дерево fiber нод, зазначених побічними ефектами. Ефекти описують роботу, яка повинна бути виконана в наступній commit фазі. Під час цієї фази React приймає дерево fiber нод, позначене ефектами, і застосовує його до інстанси. Він переглядає список ефектів і виконує поновлення DOM і інші зміни, видимі користувачеві.

Важливо розуміти, що робота під час першого рендеру може виконуватися асинхронно. React може обробити один або кілька fiber нод в залежності від доступного часу, потім зупинитися, щоб заховати виконану роботу і поступитися якоїсь події. Потім він продовжує з того місця, на якому зупинився. Іноді, однак, може знадобитися відмовитися від виконаної роботи і почати все спочатку. Ці паузи стали можливими завдяки тому, що робота, виконана на цьому етапі, не приводить до яких-небудь видимих змін для користувача, таким як оновлення DOM. Навпаки, наступна commit фаза завжди синхронна. Це пов'язано з тим, що робота, виконувана на цьому етапі, призводить до змін які в свою чергу є видимі користувачеві, наприклад, обновлення DOM. Ось чому React повинен зробити це за один прохід.

Алгоритм звірки завжди починається з самої верхньої fiber Ноди HostRoot за допомогою функції renderRoot. [6] Однак, React пропускає вже оброблені fiber Ноди, поки не виявляє ноду з незавершеною роботою. Наприклад, якщо ви викликаєте setState глибоко в дереві компонентів, React буде починатися зверху, але швидко пропускати батьків, поки не добереться до компонента, для якого був викликаний метод setState.

Коли React обходить дерево fiber нод, він використовує цю змінну, щоб дізнатися, чи є ще fiber нода з незавершеною роботою. Після обробки поточної fiber Ноди змінна буде або містити посилання на наступну ноду в дереві, або містити null. В цьому випадку React виходить з робочого циклу і готовий коммітов зміни.

1.10 Висновки

SPA - це підхід до веб-розробки, в якому знаходиться вся програма розміщені на одній сторінці і у користувача виникає враження ніби він використовує настільну програму.

У SPA не оновлюються повноцінні сторінки після завантаження програми. Натомість логіка попереднього висловлення завантажується вперед і подається з точки зору обміну поданнями в межах регіонів вмісту.

Структури MV * забезпечують механізм, що використовується SPA для з'єднання даних з наших запити сервера з поданнями, які користувач бачить та взаємодіє з ними.

Модулі забезпечують інкапсуляцію стану та / або даних. Вони також допомагають коду залишатися розв'язаним та простішим у обслуговуванні.

Здійснено аналіз найбільш популярних SPA фреймворків. Та обрано найбільш ефективний фреймворк. Описано алгоритм fiber фрейморка React для отримання розуміння при створенні системи програмного забезпечення, а саме односторінкового застосунку та досягнення мети дисертації.

2. ПРЕДМЕТНА ОБЛАСТЬ

2.1 Музичні сервіси

Епоха музичних носіїв, а можливо, і музичних альбомів відходить у минуле. На зміну їй прийшла музика як потік і як ефір. Одночасно з нашими звичками змінюється і індустрія.

Варіантів може бути маса. Можна наштовхнутися в стрічці Facebook на альбом, Apple Music, Zvooq або Spotify. Можна включити улюблену пісню на YouTube і потім півгодини слухати related tracks, підібрані алгоритмом. Або слухати все підряд на Soundcloud Boiler Room. Майже всі ці варіанти являють собою стрімінг. [10]

Рекомендації можуть створити власне музичне оточення під середньостатистичний смак. Але вони працюють на ефективне досягнення мети, а значить, і будуть створювати такий музичний фон, який не буде навантажувати зайвими думками або відволікати від справ.

Доступного і безкоштовного контенту стало так багато, що кінцевий слухач в ньому майже тоне.[9] Здається, що тепер самі артисти повинні платити аудиторії, аби їх тільки почули. Замість цього вони почали вкладатися в маркетологів, відео та SMM-менеджерів і заробляти на живих виступах - все-таки заробіток з окремого прослуховування на Стрімі виявився досить прибутковим.

2.2 Технічні показники стрімінг сервісів

Суть стрімінга музики полягає в тому, що вам не потрібні не тільки фізичні носії, але навіть завантажувати файли більше не потрібно на комп'ютер або телефон. Все, що потрібно для прослуховування музики це працює інтернет-з'єднання. Величезна перевага будь-яких стрімінгових сервісів музики полягає у

величезній кількості музики, яку вони пропонують вам послухати прямо зараз. Будь-трек, який ви коли-небудь десь чули доступний для вас прямо тут і зараз.

Більш того, стрімінговие музичні сервіси автоматично підбирають для вас все нову і нову музику. Тому будь-які стрімінговие сервіси самостійно намагаються вибрати саме ту музику, тих виконавців, які вам сподобаються. Алгоритми добірки не ідеальні, але вони здатні тримати нову музику в рамках ваших улюблених жанрів і намагатися підбирати її від виконавців, які схожі на тих, яких ви так любите слухати.

На перший погляд різниці в якості звучання музики немає, але це тільки на перший погляд, різні стрімінговие сервіси пропонують різну якість звучання, причому, іноді ця різниця може бути колосальна.

Але тут треба розуміти нюанси якості відтворення музики. Наприклад, якщо ви слухаєте її тільки під час поїздок в громадському транспорті або під час прогулянки по місту, якість звучання може відійти на другий план. [9] Але якщо ви любите слухати улюблені композиції в тиші, вдома або в офісі, тут, звичайно, дуже важливо, щоб посилається до вас сигнал був максимально можливої якості.

Також треба враховувати, що різні сервіси використовують різні формати зберігання музичних файлів, з різним бітрейтом і динамічним діапазоном. Наприклад, Spotify використовує формат Ogg Vorbis, а Deezer mp3. При однаковому бітрейті файли Ogg Vorbis звучать трохи краще, ніж mp3. Але при цьому ж, якщо оплатити максимальний тариф в Deezer ви зможете слухати вже FLAC записи в CD якості, в той час, як у Spotify немає такої можливості. Саме тому важливо розуміти який сервіс і за яких умов здатний задовольнити ваші запити.

Кожен стрімінговий сервіс намагається зробити якість програється музики максимально хорошим, але кожен сервіс робить це відповідно до своїх можливостей і запитам цільової аудиторії. Тому якщо звернутися до технічних особливостей зберігання і передачі музики ми побачимо, що сервіси відрізняються один від одного. Один сервіс зберігає музику тільки в стислому

вигляді, наприклад в MP3, інший зберігає її в CD якості, а третій здатний надати вам доступ до Hi-Res записів.

У табл. 2.1 навереді формати які кожен стрімінговий сервіс музики зберігає дані і як змінюється якість звучання музики в залежності від обраного тарифного плану.

Таблиця 2.1 — Формати стрімінговиї сервісів

Стрімінговий сервіс	Максимальна якість для мобільних клієнтів (Кб / Сек)	Максимальна якість для комп'ютерів (Кб / Сек)	Підтримувані формати файлів
Spotify	320	320	Ogg Vorbic
Google play Music	320	320	MP3, ACC,WMA,FLAC,Ogg Vorbic,ALAC
Apple Music	256	256	AAC
Tidal	320	1411	FLAC,ALAC,AAC
Amazon Music	256	256	MP3
SoundCloud	64	64	Opus
Pandora	192	192	AAC
Deezer	320	1411	MP3,FLAC
Qoubuz	320	4608	MP3,FLAC

Дані з таблиці вище корисні тим користувачам, які вже знайомі з методами та форматами зберігання музики в цифровому вигляді.

Якщо слухати музику за допомогою стрімінгових сервісів найчастіше в дорозі, на прогулянці або в громадському транспорті, MP3 з бітрейтом 320 Кб /

Сек буде для вас абсолютно достатнім. Якщо слухати музику вдома в тихій обстановці на хорошому обладнанні, тоді вам необхідний формат FLAC або MQA

На рис. 2.1 можна спостерігати графік бітрейду.

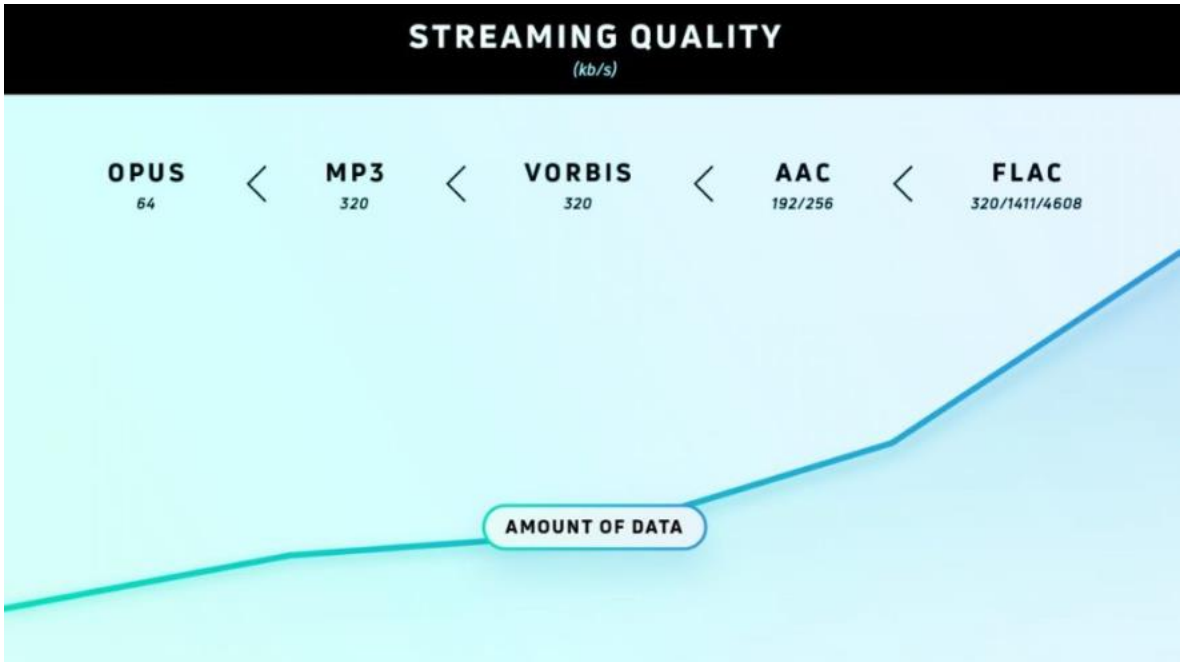


Рисунок 2.1 – Графік якості музичних файлів

Як видно з графіка найнижча якість у MP3 файлів навіть при бітрейді в 320 Кб / Сек, це пов'язано з особливостями роботи алгоритму стиснення музики. У формату AAC якість вище навіть при більш низькому бітрейті, а найвища якість у музики, що зберігається і передається в форматі FLAC і MQA.

2.3 Аналіз інших сервісів

Apple Music

Сервіс запущений в 2015 році, але йому вистачило всього декількох років, щоб обійти короля стрімінга - Spotify - по числу користувачів.

Слухати Apple Music тепер можна і через підписку Apple One. Вона включає в себе чотири сервіси: Apple Music, Apple TV +, Apple Arcade, iCloud. Виберіть план для себе або для всієї родини. При цьому у кожного буде індивідуальний

доступ до всіх сервісів на всіх особистих пристроях. Графічний інтерфейс виконаний у мінімалістичному форматі на рис. 2.2.

Три перші місяці можна користуватися безкоштовною пробною версією, потім.

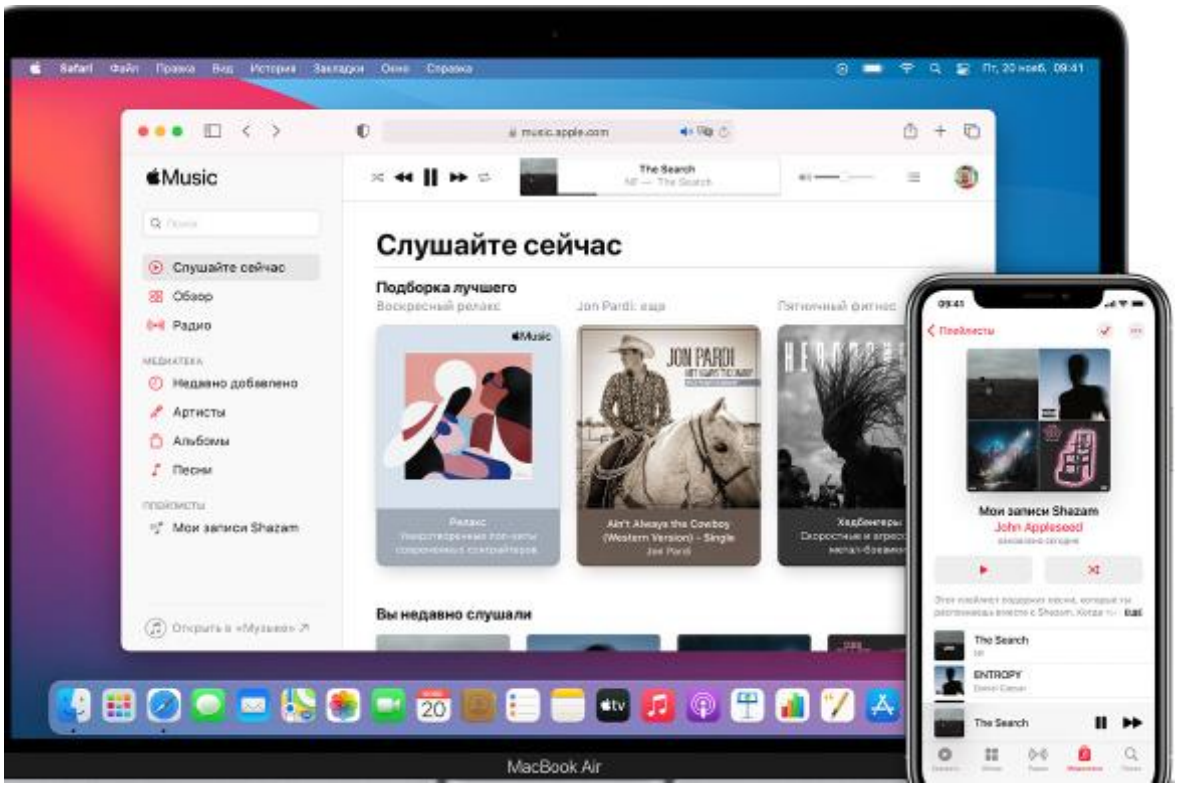


Рисунок 2.2 – Інтерфейс Apple Music

Плюси :

- понад 70 млн треків (заявлено в каталозі);
- власний аудіоформат Apple AAC з бітрейтом 256 кбіт / с. За якістю відповідає 320 кбіт / с кодека MP3;
- деякі композиції певних виконавців бувають доступні виключно в Apple Music.
- є цензура - без спеціального дозволу в налаштуваннях ти не зможеш слухати пісні, в яких є "недруковані" слова;
- функція соцмережі "Коннект" - дозволяє стежити за новинами артистів і кураторів, ділитися з друзями улюбленою музикою;
- можливість завантажувати треки на пристрій і слухати оффлайн;
- автоматичні добірки нових перспективних виконавців улюбленого жанру;

- відеокліпи хітів різних десятиліть і сучасних виконавців;
- плейлисти на основі призначених для користувача переваг;
- відсутність реклами;
- графічний інтерфейс
- швидкодія
- можна читати тексти пісень з випередженням або синхронно з прослуховуванням - режим «Текст» включений за замовчуванням, поки ви його не вимкнете;
- можно користуватися всіма можливостями Apple Music в машині за допомогою CarPlay;
- можливість синхронізації з Apple Watch - можна слухати музику прямо з годин.

Мінуси:

- немає веб-версії, потрібно ставити iTunes, який досить важкий і в деяких випадках може сильно уповільнювати роботу пристроїв не від Apple;
- незручний еквалайзер в мобільній версії, мало налаштувань.

Користувачам девайсів від Apple Music. Цей музичний потоковий сервіс ідеально поєднується і синхронізується з усіма яблучними пристроями.

YouTube Premium

Універсальний стрімінговий сервіс від компанії Google, створений в результаті злиття декількох програм - Google Play Музики, YouTube Music і YouTube Red. Зовнішній вигляд зображено на рис. 2.3.

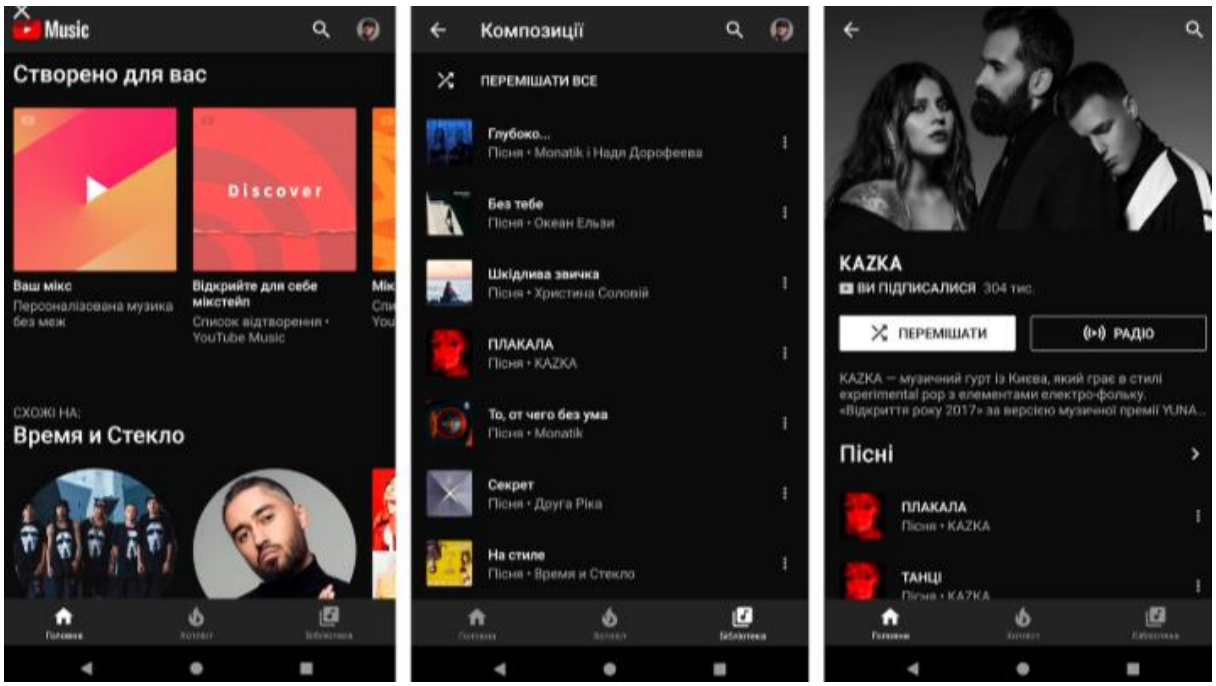


Рисунок 2.3 – Інтерфейс YouTube Premium

Користувачам ОС Android досить просто зайти в додаток, яке є на пристрої за замовчуванням, і оформити підписку.

Плюси:

- велика колекція музики;
- перегляд будь-яких відео-роликів на платформі YouTube без реклами;
- доступ до ексклюзивних серіалів і повнометражним картинам;
- можливість зберігати аудіо і відео через додаток YouTube, щоб дивитися або слухати їх оффлайн. Скачаний контент буде доступний протягом місяця;
- фонове відтворення музики, роликів та фільмів;
- простий доступ з будь-яких пристроїв і операційних систем;
- створення автоматичних добірок в рамках улюблених жанрів користувача;

Мінуси:

- робота алгоритмів автоматичної підбірки композицій поки залишає бажати кращого;

Spotify

Зроблено в Швеції. Один з лідерів в ніші музичного стрімінга як і раніше залишається одним з кращих сервісів за багатьма показниками. З липня 2020

Spotify доступний на території Росії, України, Білорусії і Казахстану. Раніше, щоб зареєструватися в ньому, потрібно було прикидатися іноземцем за допомогою мережі VPN. Зовнішній вигляд зображено на рис. 2.4.

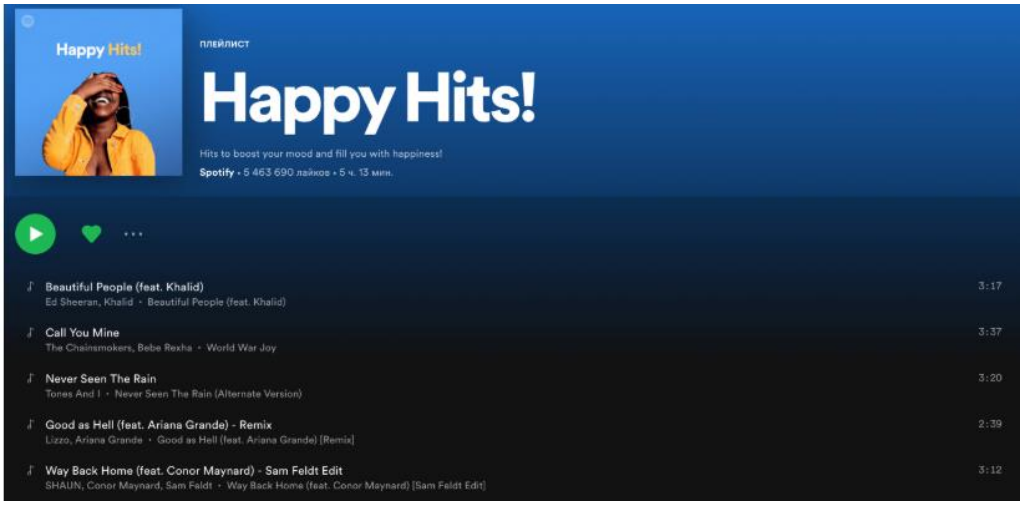


Рисунок 2.4 – Інтерфейс Spotify

Використовувати сервіс без оплати можна безстроково, але з обмеженнями: невисокий бітрейт в 160 Кбіт / с, реклама, мало кураторських плейлистів. У мобільному додатку можна завантажувати треки на смартфон, вибирати конкретну композицію в списку, а перескочити можна тільки через 6 пісень за годину.

Плюси:

- медіатека понад 50 млн пісень;
- гарна якість звуку: в преміум-доступ 320 Кбіт / с у форматі Ogg Vorbis (вище, ніж MP3);
- для деяких пісень доступні тексти і розповіді про створення треків від самих артистів (за підтримки сайту genius.com);
- дуже точні алгоритми підбору композицій по перевагах користувача;
- інтеграція з PlayStation, соцмережею Facebook, сервісами SoundHound і Shazam;
- сервіс спритно працює з будь-якими ОС і практично всіма пристроями;
- функціональність повноцінної музичної соцмережі;
- доступ до прослуховування музики онлайн в безкоштовному режимі;

- можливість зберігати треки на телефоні і слухати без інтернет-з'єднання;
- мільйони кураторських плейлистів, які створюються кожен день, і щотижневий Discover Weekly - ретельно опрацьована добірка нових виконавців за смаками користувача від музичних експертів;
- можна ділитися своєю колекцією з друзями, бачити, що слухають вони, і створювати спільні листи;

Мінуси:

- Веб версія додатку має не дуже повільну швидкодію;
- Швидкість роботи веб версії;
- Інтерфейс користувача спроектований у веб версії дуже погано;

SoundCloud

Це не просто стрімінговий сервіс, а справжня музична соціальна мережа. SoundCloud - швидше за хмару для зберігання аудіозаписів і зручний майданчик для дистрибуції власної музики. Практично кожен інтернет-користувач хоч раз так стикався з SoundCloud. Цей сервіс - як YouTube, тільки не в відео-, а в аудіо-сфері. Зовнішній вигляд зображено на рис. 2.5.

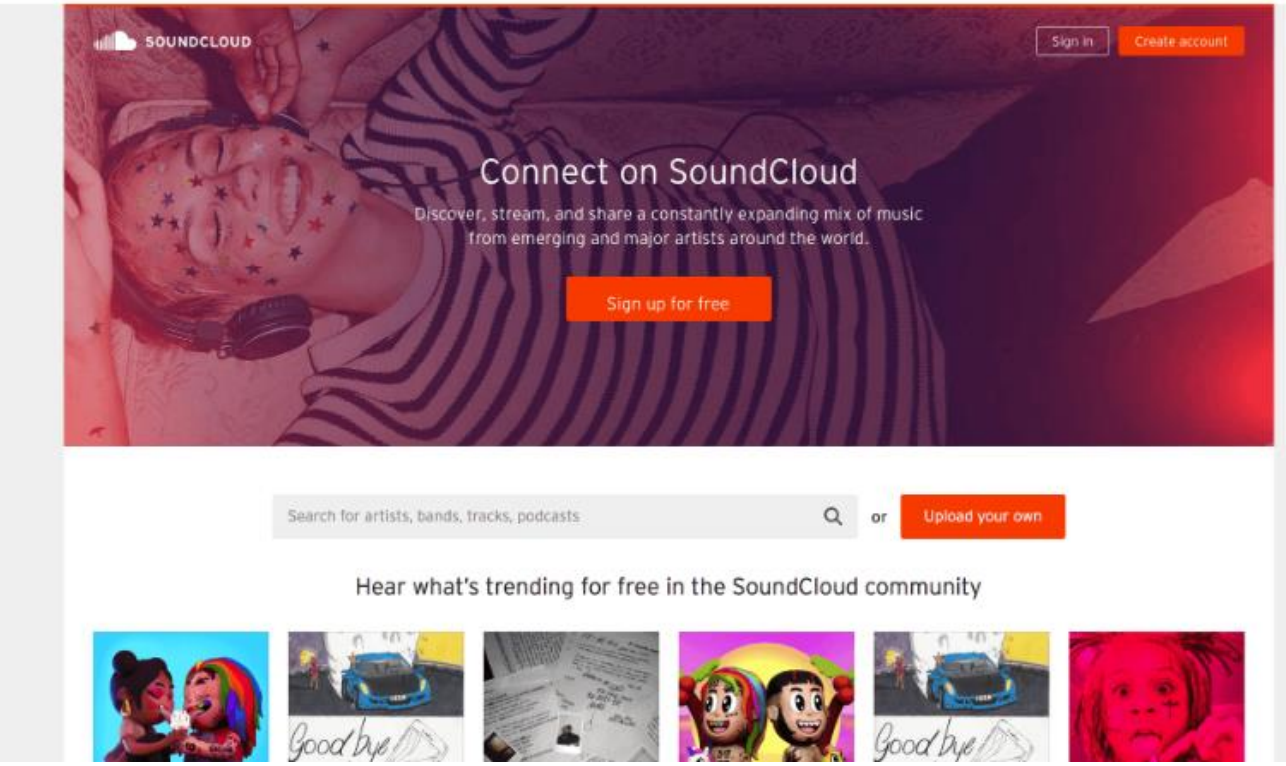


Рисунок 2.5 – Інтерфейс SoundCloud

Плюси SoundCloud:

- велика колекція не тільки музики, а й різних подкастів, радіопрограм, звуків і всяких цікавих авторських композицій - бібліотека складається з більш ніж 125 млн аудіозаписів;
- хмарне сховище для завантаження своїх аудіо-файлів, яких немає в медіатеці сервісу;
- одна з найпопулярніших платформ для дистрибуції музики: можна викладати власні твори і отримувати коментарі, відгуки та рекомендації, а також записувати свої аудіо в режимі онлайн;
- деякі популярні бенди публікують на SoundCloud нові треки до їх офіційного випуску ;
- опція розміщення плеєра зі своїм аудіо на інших ресурсах;
- добре налагоджена веб-версія, додатки для iOS і Android, інтеграція з ігровою приставкою Xbox One, аудіосистемою Sonos і плеєром Chromecast;
- користувачі можуть створювати спільноти, підписуватися на інших людей і обрані групи;
- реальна можливість контактувати з улюбленими виконавцями та кумирами;
- зручний, інтуїтивно зрозумілий інтерфейс;

Мінуси:

- не дуже багата фонотека популярної музики;
- далеко не всі авторські треки викладаються в якісному виконанні;
- сервіс грунтувався як платформа для артистів і продовжує розвиватися в першу чергу з урахуванням їх інтересів

2.4 Висновки

В даному розділі досліджено то описано предметну область. Предметом дослідження є реалізація односторінкового застосунку в музичній сфері. Для створення актуальної системи було проаналізовано всіх можливих конкурентів, їх сильні та слабкі сторони.

Це дає змогу розробити такий графічний інтерфейс, який відповідає ввимогам розробки, орієнтована на користувача. За допомогою аналізу інших схожих ресурсів було розроблено власний інтерфейс користувача. Завдяки цьому розроблений інтерфейс позбавлений критичних мінусів, які були присутні в подібних системах.

Також в ході розробки власного інтерфейсу було виділено основні властивості:

- Природність інтерфейсу
- Узгодженість інтерфейсу
- Узгодженість в межах продукту
- Узгодженість в межах робочого середовища
- Дружелюбність інтерфейсу (принцип «пробачення» користувача)
- Принцип «зворотного зв'язку»
- Простота інтерфейсу
- Гнучкість інтерфейсу

3 РЕАЛІЗАЦІЯ СИСТЕМИ

3.1 Вибір технологій

React - популярна технологія для Frontend-розробки, але це тільки View (Представлення) шар, а значить, у нас в розпорядженні тільки V від MVC - Model-View-Controller (Модель - Представленіе- Контролер) архітектури. React часто згадується серед інших фреймворків, але все, що він дає - це View.

React - це мова шаблонів і всього кілька функцій-hook'ов для рендеринга HTML. Так як він заснований на компонентному підході, ви можете розробляти додаток с допомогою React просто вказуючи, як би ви хотіли бачити той чи інший елемент. React буде автоматично оновлювати елемент, коли лежать в його основі дані зміняться[15].

Головні принципи React: гнучкість, ефективність і декларативний код. Так як React гнучкий, то можна використовувати один і той же код в декількох проектах, створювати на його основі нові додатки і навіть використовувати в уже існуючій базі код без переробок.

Flux, шаблон проектування, відповідає M в MVC. Це архітектура, відповідальна за створення шару даних в JavaScript додатках і розробку серверної сторони в веб-додатках. Flux доповнює складові компоненти виду View в React, використовуючи односпрямований потік даних[21].

Також можна сказати, що Flux більше ніж шаблон, більше ніж фреймворк і має 4 головних компонента (більш докладно вони будуть розглянуті пізніше):

- Диспетчер (Dispatcher)
- Сховище (Stores)
- Вистави (Views) (React компонент)
- Дія (Action)

Це не схоже на звичний MVC, який ми звикли бачити в інших фреймворків. Дійсно, там є контролер, але здебільшого це контролер, який відповідає за Views.

Views знаходяться в самій вершині ієрархії, і вони передають функціонал і дані елементам-нащадкам.

Flux слід концепції односпрямованого потоку даних, що робить його простим для пошуку помилок так як можна швидко знайти проблемне місце. Дані проходять через прямий потік в програмі[21].. React і Flux - на даний момент два найпопулярніших фреймворка, які використовують принцип односпрямованого потоку даних.

У той час як React використовує віртуальний DOM для відображення змін, Flux робить це трохи інакше. У Flux, взаємодія з призначеним для користувача інтерфейсом викличе ряд дій, які можуть змінити дані програми.

Flux має відкритий вихідний код, і це скоріше шаблон проектування, а не фреймворк, тому його можна використовувати відразу. Те, що відрізняє його від інших фреймворків, то відрізняє його і від шаблону проектування MVC.

Flux допомагає зробити код більш передбачуваним, в порівнянні з MVC фреймворками за рахунок однонаправленого потоку. При розробці система не виникало складних взаємодій між джерелами даних.

Flux вигідно відрізняється більш організованим потоком даних - односпрямованим. Те, що він односпрямований, головна особливість Flux. Ці дії поширюються на нову систему щодо взаємодії з користувачем.

The Flow - Flux дуже вимогливий до потоку даних в додатку. Dispatcher даних встановлює та виконує суворі правила і виключення для управління потоком. У MVC немає такої речі, і потоки реалізуються по-різному.

Односпрямований потік в Flux - в той час як MVC двонаправлений в своєму потоці, під Flux всі зміни проходять через один напрямок, через Dispatcher даних. Store не може бути змінено саме по собі, і той же самий принцип працює для інших Actions[21]. Зміни, які необхідно внести, повинні пройти через Dispatcher, через Actions.

Store - в той час як MVC не може моделювати окремі об'єкти, Flux може робити це для того, щоб зберігати будь-які пов'язані з додатком дані.

Коли постає питання про те, що вибрати Flux або MVC, краще вибрати Flux, тому що його простіше зрозуміти і працювати з мінімальним використанням коду. Flux дозволяє вам ефективно структурувати ваш додаток, тому що мова програмування React інтегрована з величезною базою коду і жахливої ран-тайм складністю.

Можливо, для того, щоб зрозуміти, чому односпрямований потік даних є найкращим рішення, варто дізнатися про мінуси двонаправленого потоку даних.

У двунаправленном потоці даних є стандартний потік даних - Model-View-Controller. Однак, коли додатки стають більш складними на Controller лягає дуже велике навантаження.

Controller бере на себе величезну відповідальність, як за збереження стану програми, так і за дані. Крім того, каскадні оновлення роблять додатки складним для розуміння і пошуку помилок. В кінцевому підсумку ми маємо додаток, результати роботи якого непередбачувані.

В однобічному потоці даних ця проблема пом'якшується, і таким чином досягається передбачуване стан додатки. Коли потік даних Односпрямована, зміни в шарі View тягнуть за собою зміни в шарі даних. Пізніше ці зміни будуть відображені в View. Проте, View не впливає безпосередньо на дані програми.

Компоненти архітектури Flux взаємодіють між собою скоріше як EventBus і менш, ніж MVC[11].

Як згадувалося раніше, Flux це не справжня бібліотека або фреймворк, це новий вид архітектури, яку Facebook використовує для роботи з React зсередини. Отже, основна функція Flux буде доповнити React і реалізувати односпрямований потік даних.

Контролери в MVC і Flux розрізняються. Тут контролери є Controller-View і знаходяться на самій вершині ієрархії. View - це React компоненти.

Весь функціонал, як правило, знаходиться в Store. В Store виконується вся робота і повідомляється Dispatcher, які події / дії він прослуховує. На рис. 3.1 зображено схематично потік даних. [21]

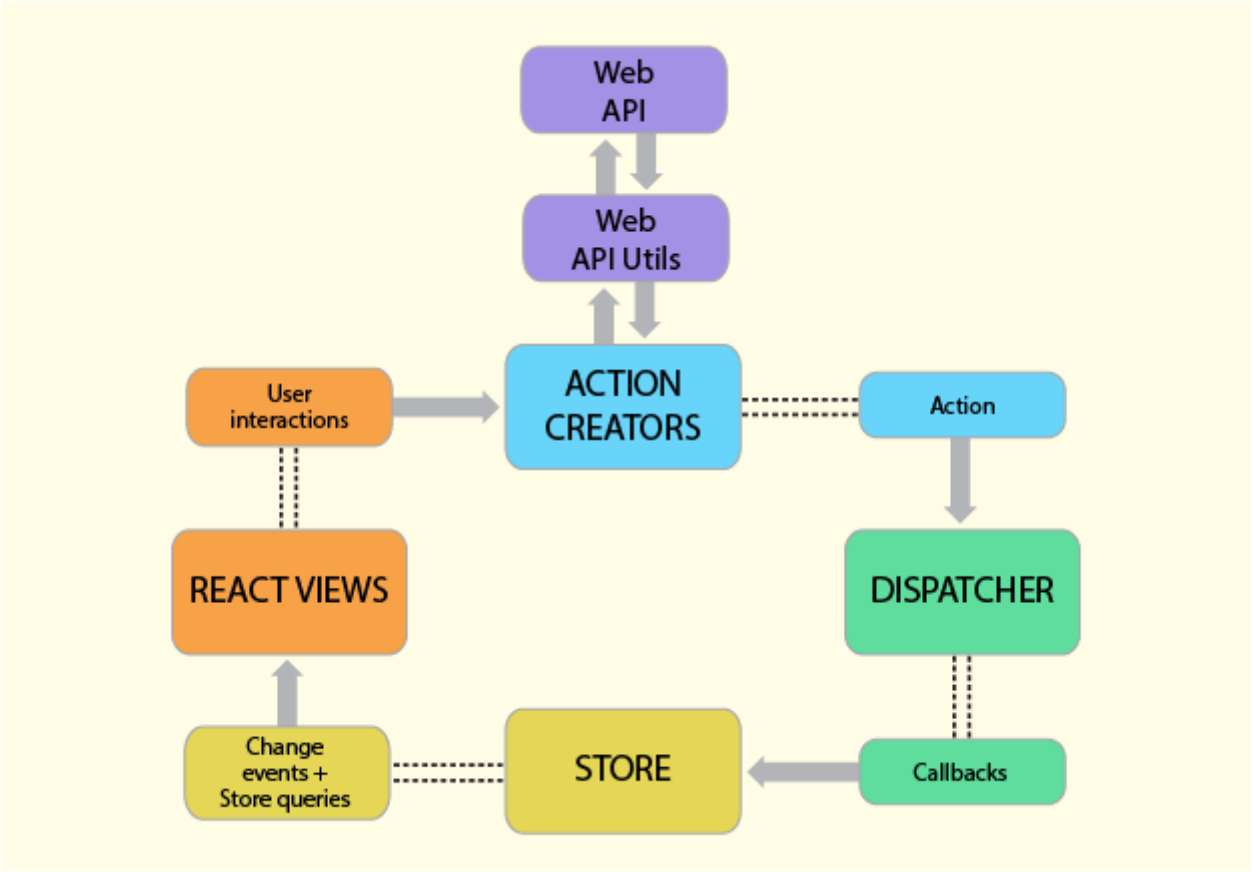


Рисунок 3.1 – Архітектура Flux

У стандартній архітектурі Flux наступні компоненти:

- Actions - помічники, які передають дані в Dispatcher
- Dispatcher - отримує ці дії і передає корисне навантаження зареєстрованим callback-му.
- Stores - діють як контейнери для стану програми та логіки. Реальна робота додатка відбувається в Stores. Stores, зареєстровані для прослуховування дій Dispatcher, будуть відповідно і оновлювати View.
- Controller Views - React компоненти захоплюють стан з Stores, а потім передають дочірнім компонентів.

Коли відбувається подія, Dispatcher відправляє "корисне навантаження" в Store, який зареєстрований для прослуховування конкретно цієї події. Тепер в Store необхідно оновити View, що в свою чергу викликає дію. [19] Дія, точно також як і ім'я, тип події та багато іншого відомі заздалегідь.

View поширює Action через центральний Dispatcher, і це буде відправлено в різні Stores. Ці Stores будуть відображати бізнес-логіку програми та інші дані. Store оновлює все View. [13]

Найбільш добре це працює спільно зі стилем програмування React, тоді Store відправляє поновлення без необхідності детально описувати, як змінювати уявлення між станами.

Це доводить, що шаблон проектування Flux слід за односпрямованим потоком даних. Action, Dispatcher, Store і View - незалежні вузли з конкретними вхідними та вихідними даними. Дані проходять через Dispatcher який зображено на рис. 3.2, центральний хаб, який в свою чергу управляє всіма даними[13] .

Dispatcher діє як реєстр із зареєстрованими callback-ами, на які відповідають Store. Store будуть давати зміни, які обрані Controller-Views.

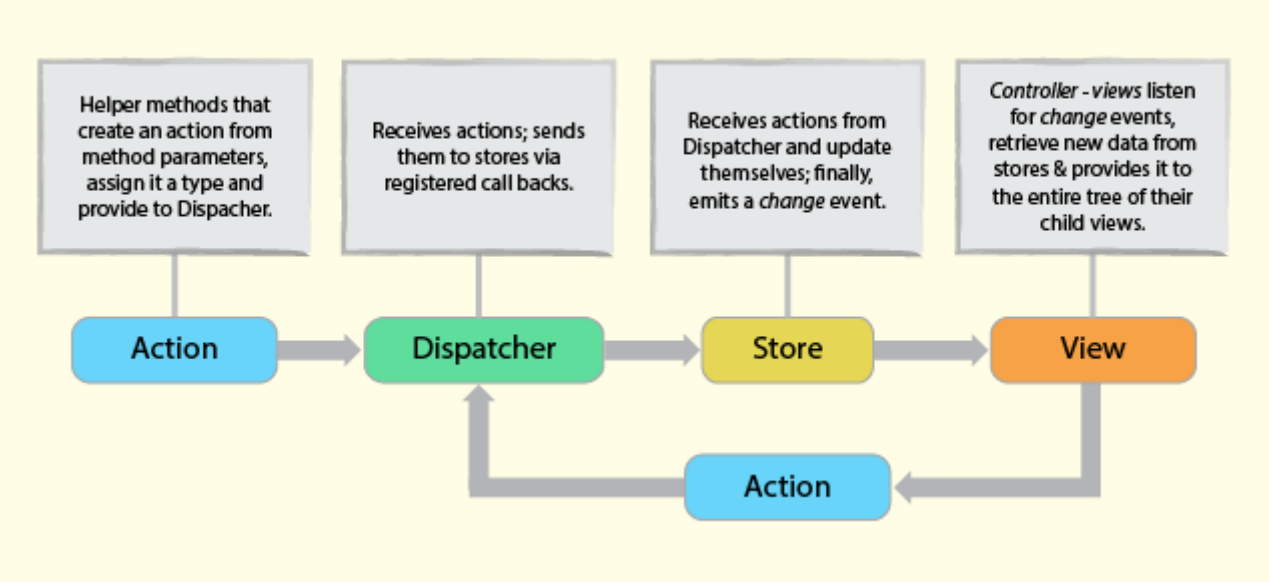


Рисунок 3.2 – Потік даних

Це відбувається, коли View поширюється в системі

Це доводить, що в Flux немає двосторонніх прив'язок (bind), що структура на кшталт функціональному відносного програмування, і ще щось на зразок потокового програмування.

Залежності, які відбуваються в Store, зберігаються в суворій ієрархії, тоді як Dispatcher обробляє оновлення. Ця структура також вирішує проблеми, які природним чином виникають при двосторонньої прив'язці.

Redux це- згідно GitHub, Redux передбачуваний State контейнер для JavaScript додатків. Велика частина концепцій аналогічні функціональним програмування, а всі дані зберігаються в одному Store.

Незалежно від розміру програми, Redux завжди є єдиним об'єктом, на відміну від Flux, який містить різні сховища для різних об'єктів. Якщо в даних були зроблені зміни, це не впливає безпосередньо на стан. Таким чином, state є незмінним. [21]

Всі оновлення і маніпуляції виконуються на дереві станів. Але це не робить додаток повільним, так як даний можуть спільно використовуватися різними версіями дерева станів. [11]

Оновлення стану програми виконуються через Actions, які представляють собою прості об'єкти, але містять властивість типу, що зображує вид виконаних дій. Також сюди включені дані, які описують дії.

Dispatcher сховища відправить дію і звідти воно відправиться в Reducer, а потім в поточний древо станів. Тут будуть дії програми будуть описані - різні речі, які додаток може робити. Всього лише одного Reducer'a буде досить для трансформації станів і дій.

3.2 Основні компоненти

Основні компоненти які використовують в різних частинах системи. Для зручності користування і дотримання архітектурної структури в файловіх системі проекту було створено окрему папку де будуть розміщенні основні компоненти.

Назва папки components і в ній знаходяться основні компоненти. Для візуалізації компонентів у табл. 3.1 використовується підхід styled-components.

Таблиця 3.1 – основні компоненти.

album-cover	Даний компонент в якості пропсів приймає значення (інформацію про конкретний елемен, висоти, розміру) і

	відображає цей елемент на основі отриманних пропсів. Для покращення і збільшення швидкодії компонент обертається в HOK React.memo щоб запобігти непотрібного ререндеру.
app-footer	Даний компонент існує для відображенняя нижньої частини додатку. Складається з 3 частин (AppFooterWrapper,FooterLeft,FooterRight) які імпортуються з корневої папки і описані у вигляді styled-components. В якості проспів приймає картинки і силки необхідні для відображення. Також обернутий в HOK React.memo.
app-header	Компонент який відображає навігацію по додатку. В якості пропсів приймає значення силок і в динамічно розприділяє їх по навігації. Також являється складним компонентів і складається з інших компонентів (AppHeaderWrapper,HeaderLeft,HeaderRight). Також в даному компоненті я елемент який виконую роль пошуку по системі певну інформацію, він має окрему логіку а, для стилізації компоненту використано готовий компонент із бібліотеки antd. Даний компонент також обернутий в HOC React.memo.
pagination	Компонент який відповідає за пагінацію і маю свою логіку використання яку здійснює за допомогою функції itemRender(props) яка приймає певні пропси і відповідає за інтерфейсом переключення. Для стилізації використано компонента із бібліотеки antd , а контейнер описано в styled-components. Так як даний елемент буде часто використовуватися для збільшення продуктивності також обернутий в

	компонент React.memo.
radio-ranking-cover	Компонент який відповідає за відображення рейтингу для радіо. В якості вхідних пропсів отримує дані необхідні для відображення компонента, а саме інформація про радіо. Також обернутий в компонент React.memo.
radio-recommend-cover	Компонент який відображає рекомендовані елементи а саме рекомендовані альбоми або пісні. Також являється компонентом який часто викростовується тому обернутий в React.memo. Дані які необхідно відобразити отримую в пропсах. А візуальні дані описані в styled-components.
song-operation-bar	Компонент який відповідає за інтерфейс пластинки або альбому. Вхідні дані поступають в пропсах і деструктивним синтаксисом JS отримуються необхідні дані, які розприділяються по компоненту. Як і інші компоненти також обернутий в React.memo та стилізація задана в вигляді styled-components.
theme-cover	Компонент який відображає інформацію про окрему пісню виконавця. В якості пропсів приймає дані про обкладинку пісні та його виконавця і також про кількість прослуховувань даної пісні. Обернутий в React.memo для збільшення продуктивності. Для стилізаціх використовується styled-components.
theme-header-player	Компонент який є частиною реалізації плейера. Відображає інформацію про вибрану пісню. На основі вхідних пропсів.
theme-header-small	Компонент який є частиною інших компонентів і використовується для відображення тайтлів альбомів чи

	виконавця. В залежності від вхідних пропсів може відображати імя автора та силку на самого автора
theme-header-song	Компонент який відповідає за відображення альбом певного виконавця. Відображає всю інформацію про даний альбом. Отримує інформацію з Redux store за допопомогою UseSelector. Обернутий в компонент React.memo.
top-ranking	Компонент який відображає топ чарт. Топ чарт включає в себе інформацію про найбільш найактуальніші пісні. На вході компонент отримую вхідні пропси з яких деструктивним методом отримується необхідна інформацію, а саме інформація для кожної пісні яка входить. Також в компоненті наявна кнопка для добавлення пісні в список програвача. Перед добавленням пісні в програвач викликається useDispacth, який перед добавленням пісні спочатку надішле інформацію в Redux store.

3.2 Побудова системи управління

Стор (Store) - це об'єкт, який з'єднує ці частини разом. Стор бере на себе такі завдання:

- Містить стан додатки (application state)
- Надає доступ до стану за допомогою getState ()
- Може випускати оновлення стану за допомогою dispatch (action); Обробляє скасування реєстрації слухачів за допомогою функції, що повертається subscribe (listener).

Стан - це дані, з якими працює ваш компонент (компоненти) - він містить дані, необхідні компоненту, і він диктує, що компонент відображає. Як тільки

об'єкт стану змінюється, компонент рендерінг. Якщо станом програми керує Redux, тоді зменшення відбувається там, де відбуваються зміни стану.

Важливо відзначити, що буде тільки один стор. Якщо потрібно розділити логіку обробки даних, то потрібно буде використовувати композицію редюсеров (reducer composition) замість використання безлічі сторі (stores) у табл. 3.2.

Для створення стору використовується функція createStore. createStore(reducer, composeEnhancers(applyMiddleware(thunk)))

Таблиця 3.2 – Store.

reducer	Представляє собою композицію всіх редюсерів.
composeEnhancers	Кінцева функція отримана шляхом композиції переданих функцій справа наліво яка очікує як параметри, що кожна функція приймає один параметр. Її значення, що повертається буде представлено в якості аргументу для функції стоїть зліва, і так далі. Винятком є самий правий аргумент, який може приймати кілька параметрів, оскільки він буде забезпечувати підпис для отриманої в результаті функції.
applyMiddleware	Використання проміжного програмного забезпечення є підтримка асинхронних дій без великого типового коду. Це робиться, дозволяючи відправляти асинхронні дії на додаток до звичайних дій.
thunk	Зазвичай Redux-Thunk використовує для асинхронних запрошень для зовнішнього API, для отримання або збереження даних. Redux-Thunk дозволяє легко розподілити екшени, які слідують за «життєвим циклом» запрошення до зовнішнього API.

В Таблиця 3.2 – Store видно з яких компонентів складається Redux Stor. Для зручності було винесено всі редюсери в окрему папку.

Функція composeEnhancers використовується для додавання логера для комфортної роботи з redux і функції compose.

Редуктор або редюсер - це чиста функція, яка приймає стан програми та дії як аргументи та повертає новий стан. Наприклад, редуктор автентифікації може приймати початковий стан програми у вигляді порожнього об'єкта та дії, що повідомляє, що користувач увійшов у систему та повернув новий стан програми із зареєстрованим користувачем. Зміна стану на рис. 3.3.

Чисті функції - це функції, які не мають побічних ефектів і повертатимуть однакові результати, якщо будуть передані однакові аргументи.

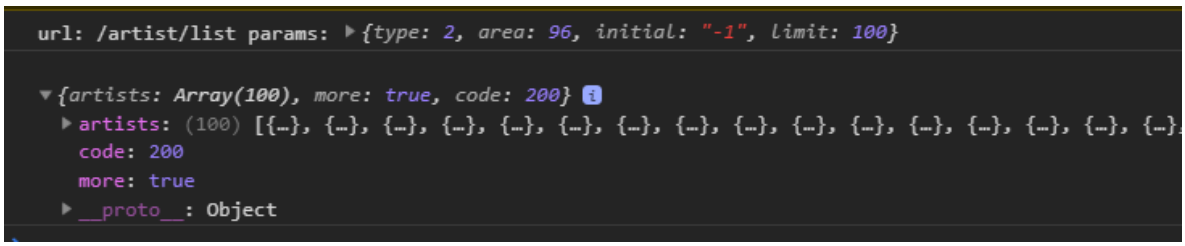


Рисунок 3.3 – Оновлення стору

На рис. 3.3 зображенно, що при будь якій діяльності звязаної з стором логер буде в браузерній консолі буде логувати про будь які зміни.

Таблиця 3.3 – Редюсери стору.

recommendReducer	Редюсер який відповідає за частину з рекомендаціями
rankingReducer	Редюсер який відповідає за частину з рейтинговими чартами
songsReducer	Редюсер який відповідає за частину з піснями
radioReducer	Редюсер який відповідає за частину з раідо
albumReducer	Редюсер який відповідає за частину з альбомами
artistReducer	Редюсер який відповідає за частину з

	артистами
playerReducer	Редюсер який відповідає за частину відтворення музики

В Таблиця 3.3 – Редюсери стору зображенно всі головні редюсори з яких складається головний стор. Всі вони підписуються на події стору і залежать від нього, графічне зображення на рис. 3.4.

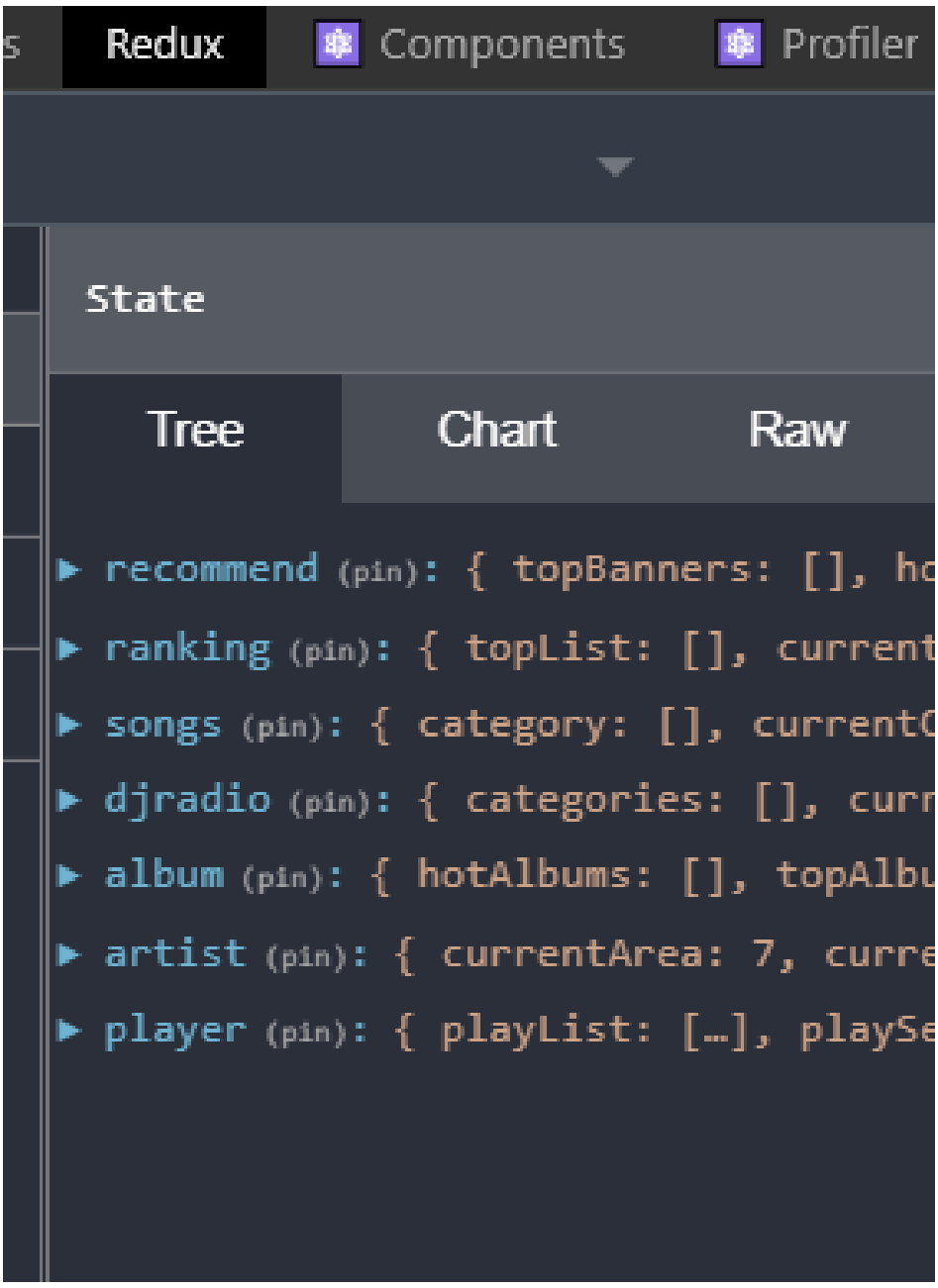


Рисунок 3.4 – Вигляд редюсерів в redux devtools

Зовнішній вигляд редюсерів в Chrome devtools. Та представлення у вигляді дерева на рис. 3.5.

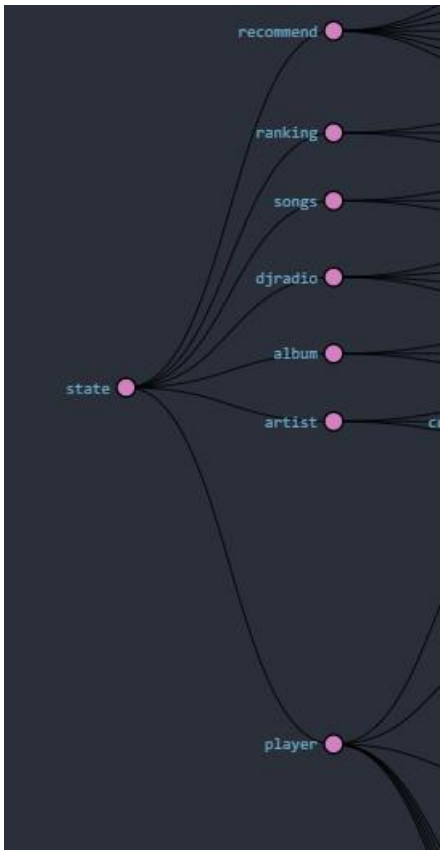


Рисунок 3.5 – Вигляд store у деревовидній структурі

Архітектура Redux обертається навколо строго односпрямованого потоку даних.

Це означає, що всі дані в додатку слідують одному паттерну життєвого циклу, роблячи логіку вашого застосування більш передбачуваною і легкою для розуміння. Також це сприяє більшій впорядкованості даних (data normalization), так що в кінцевому підсумку у вас не буде кількох ізольованих копій одних і тих же даних, які нічого не знають один про одного.

3.3 Роутінг в додатку

Для переміщенні в системі цього ви можете використовувати React Router. Тоді Redux буде джерелом правдивих даних, а React Router - єдиним джерелом URL. У більшості випадків, правильно розділяти ці поняття, але до тих пір, поки

вам не знадобиться подорожувати в часі (Time Travel) і перемотувати екшени, які змінюють URL.

React Router поставляється з компонентом <Link /> який дозволяє переміщатися по додатком. Якщо ви захочете додати деякі стилі, react-router-dom надає спеціальний <Link /> званий <NavLink />, який приймає параметри стилізації. Параметр activeStyle дозволяє застосувати стиль активного стану.

Для виконання переадресації з одного маршруту на інший в модулі react-router-dom визначено компонент Redirect.

Функція React.lazy дозволяє рендерить динамічний імпорт як звичайний компонент.

Компонент з ледачою завантаженням повинен рендери всередині компонента Suspense, який дозволяє нам показати запасне вміст (наприклад, індикатор завантаження) поки відбувається завантаження ледачого компонента.

Проп fallback приймає будь-який React-елемент, який ви хочете показати, поки відбувається завантаження компонента. Компонент Suspense можна розмістити в будь-якому місці над ледачим компонентом. Крім того, можна обернути кілька ледачих компонентів одним компонентом Suspense.

Якщо якийсь модуль не завантажується (наприклад, через збій мережі), це викличе помилку. Ви можете обробляти ці помилки для поліпшення користувацького досвіду за допомогою запобіжників. Після створення запобіжника, його можна використовувати в будь-якому місці над ледачими компонентами для відображення стану помилки.

Рішення про те, де в вашому додатку ввести поділ коду, може бути непростим. В ідеалі, слід вибрати такі місця, щоб код поділявся на бандли приблизно одного розміру, тим самим підтримуючи хороший користувальницький досвід.

Ось приклад того, як організувати поділ коду на основі маршрутів за допомогою React.lazy і таких бібліотек як React Router.Який виконую всю маршрутизацію в односторінковому застосунку а в табл. 3.4 наведено маршрути.

Таблиця 3.4 – Таблиця роутингу.

path	/	component: Discover
Redirect	/discover	
path	/discover	
path	/discover/recommend	component: Recommend
path	/discover/ranking	component: Ranking
path	/discover/songs	component: Songs
path	/discover/djradio	component:Djradio
path	/discover/artist	component: Artist
path	/discover/album	component: Album
path	/discover/player	component: Player
path	/friend	component: Friend
path	/mine	component: Mine

В Таблиця 3.4 – Таблиця роутингу показано маршрути. Часто таким зручним місцем виявляються маршрути. Більшість інтернет-користувачів звикли до затримок під час переходів між сторінками. Тому і вам може бути вигідно

повторно отрендерити всю сторінку цілком. Це не дозволить користувачам взаємодіяти з іншими елементами на сторінці, поки відбувається оновлення.

3.4 Сервіси

Таблиця 3.5 – Таблиця основних функції для запитів на сервіси.

getTopAlbums	url: "/top/album", params: { limit, offset
getHotAlbums	url: "/album/newest"
getArtistList	url = "/top/artists" params : { limit: 100 }
request	Головний запит який створює інстанст який використовують всі інші запити у своїй основі. Також містить interceptors , і хендлери для обробки помилок.
getDjRadioRecommend	url: "/dj/catelist"
getDjRadioCatelist	url: "/dj/catelist"

getDjRadios	url: "/dj/radio/hot",
	params: {
	cateId,
	limit,
	offset
	}

getSongDetail	url: "/song/detail",
	params: {
	ids
	}

getLyric	url: "/lyric",
	params: {
	id
	}

getSimiPlaylist	url: "/simi/playlist",
	params: {
	id
	}

getSimiSong	url: "/simi/song",
	params: {
	id
	}

getTopList	url: "/toplist"

getRankingList	url: "/playlist/detail",
	params: {
	id
	}

getTopBanner	url: "/banner"
getHotRecommend	url: "/personalized"
getNewAlbum	url: "/top/album", params: { limit,

	offset }

getTopList	url: "/top/list",
	params: {
	idx
	}

getArtistList	url: "/artist/list",
	params: {
	cat,
	limit
	}

getSongCategory	url: "/playlist/catlist"

getSongCategoryList	url: "/top/playlist",
	params: {
	cat,
	limit,
	offset
	}

В таблиці Таблиця 3.5 – Таблиця основних функції для запитів на сервіси і для цього використовується бібліотека Axios.

Axios це один з найпопулярніших HTTP клієнтів для браузерів. У Axios є підтримка запитів, отримання відповідей від сервера, їх трансформація і автоматична конвертація в JSON. Також в ньому є захист від XSRF.

Request функція є основною і всі інші функції використовують її інстанс для відправки запитів.

3.5 Основні сторінки

Таблиця 3.6 – Таблиця компонентів програвача.

function Player()	Головний компонент який обернутий в React.memo для зменшення рендерингу компонента. Компонент який є відображенням програвача і містить в собі всі інші компоненти необхідні для програвача.
function PlayerComment()	Компонент який відображає коментарії щодо вибраного аудіозапису в програвачу є частиною головного компоненту
function PlayerInfo()	Компонент який відображає всю інформацію про аудіозапит, а саме обкладинку, виконавця, альбом виконавця та іншу корисну інформацію. Для отримання інформаці використовує useSelector з бібліотеки react-redux для отримання інформації з головного стору. Також для використовується React.useState для зберігання інформації всередині компонента отриману через стор. Для візуалізації частин інтерфейсу використано styled-componets та antd бібліотека компонентів.
function Relevant()	Компонент який відображає схожі до поточного аудіозапису. Для отримання інформації до схожого аудіозапиту виконує запит до стору через функцію useSelector. Після отримання даних відображає їх в компоненті. Якщо користувач бажає прослухати схожий аудіозапис то виконується

	useDispatch до стору який обробляється певним рудюсером все це виконується в компоненті в середині хуку useEffect який виконує запит useDispatch.
function PlayerSongs()	Компонент який відповідає за частину про плейлист і відображає наступні аудіозаписи. Для отримання даних звертається в стор завдяки useSelector. А для отримання актуальних даних використовує хук useEffect в середині якого відбувається useDispatch до стору який оброблює редюсер.
function AppPlayList()	Головний компонент який містить в собі інші компоненти які потрібні для відображення плей листу. Обернутий в компонент React.memo для збільшення продуктивності.
function PlayList()	Компонент який потрібний для відображення списку з аудіозаписами. Інформація надходить до компонента з стора завдяки useSelector. Дані необхідні для різних частин компонента розприділяються по компоненту.
function PlayHeader()	Компонент що відображає верхню частину яка містить в собі інформацію про музиканта, назву аудіозапису та інше. Дані надходять через стор завдяки useSelector. Для збільшення швидкості обернутий в HOC React.memo.
function LyricPanel()	Компонент який відповідає за відображення аудіозапису який зараз знаходиться в активному стані.Використовується компонент

	в панелі плей листа. Використовується хук useRef Для захоплення фокусу на елементі та додання анімації.Інформація в компонент надходить завдяки useSelector. Отримана інформація поширюється в елементі перед чим записана в хук useState.
function AppPlaybar()	Компонент який займається створенням та розподіленням інших компонентів. Відображає інші вище перечислені компоненти та являється керуючим компонентом. Інформація надходить через стор завдяки useSelector так як цьому компоненту потрібно багато інформації вона записується в інші 6 шуків useState. const [isPlaying, setIsPlaying] = useState(false) const [duration, setDuration] = useState(0) const [currentTime, setCurrentTime] = useState (0) const [progress, setProgress] = useState(0) const [isChanging, setIsChanging] = useState(false) const [showPanel, setShowPanel] = useState(fa lse)

На Таблиця 3.6 – Таблиця компонентів програвача відображену його складову, описано всю логіку необхідну для програвача.

А також описано додаткові дії які виконують ці компоненти.

Таблиця 3.7 – Таблиця функцій програвача.

audioRef	Необхідно для отримання фокусу і анімації

	компонента використовує хук useRef(). І за подомогою хука useEffect () виконує свою логіку.
play	Функція яка відповідає за необхідність включення обраного аудіозапису. Обернута в хук useCallback і передається і через useDispatch в стор інформацію про подію.
timeUpdate	Функція яка відповідає за початок прослуховування коли виконунається подія на функції play.
timeEnded	Функція яка відповідає за кінець прослуховування коли виконунається подія на функції play.
sliderChange	Функція яка відповідає за перелистування вперед аудуозаписів, завдання її при події відправити useDispatch що трапилась події і далі передати її до стору. Для покращення швидкодії і запобігти перерендерам компоненту обернута в хук useCallback
sliderAfterChange	Функція яка відповідає за перелистування назад аудуозаписів, завдання її при події відправити useDispatch що трапилась події і далі передати її до стору. Для покращення швидкодії і запобігти перерендерам компоненту обернута в хук useCallback

В Таблиця 3.7 – Таблиця функцій програвача описана вся логіка компонента function AppPlaybar() який контролює дії користувача і реагує на них за

допомогою утиліт з бібліотеки react-redux, необхідної для головного стору redux який відповідає за логіку системи.

Таблиця 3.8 – Таблиця actionCreators.

changeCurrentSongAction	Змінити поточний аудіозапису
changeCurrentSongIndexAction	Змінити індекс поточного аудіозапису
changePlayListAction	Змінити поточний плей лист
changeLyricsAction	Змінити плей лист
changeSimiPlaylistAction	Отримати дані про плей листа виконавця
changeSimiSongsAction	Отримати дані про виконавця
changePlaySongAction	Змінити аудіозапит
getSongDetailAction	Отримати деталі до аудіозапиту
getSimiPlaylistAction	Отримати дані поточного плей листа
getSimiSongAction	Отримати дані поточного аудіозапиту

В Таблиця 3.8 – Таблиця actionCreators відображено всі дії які створюється на окремі useDispatch показано на рис. 3.6.

Ці action відправляються в редюсер який їх оброблює і змінює стор.

Рисунок 3.6 – Зміна дій

На рис. 3.7 зображено actionCreators які завдяки useDispatch надають інформації стору.

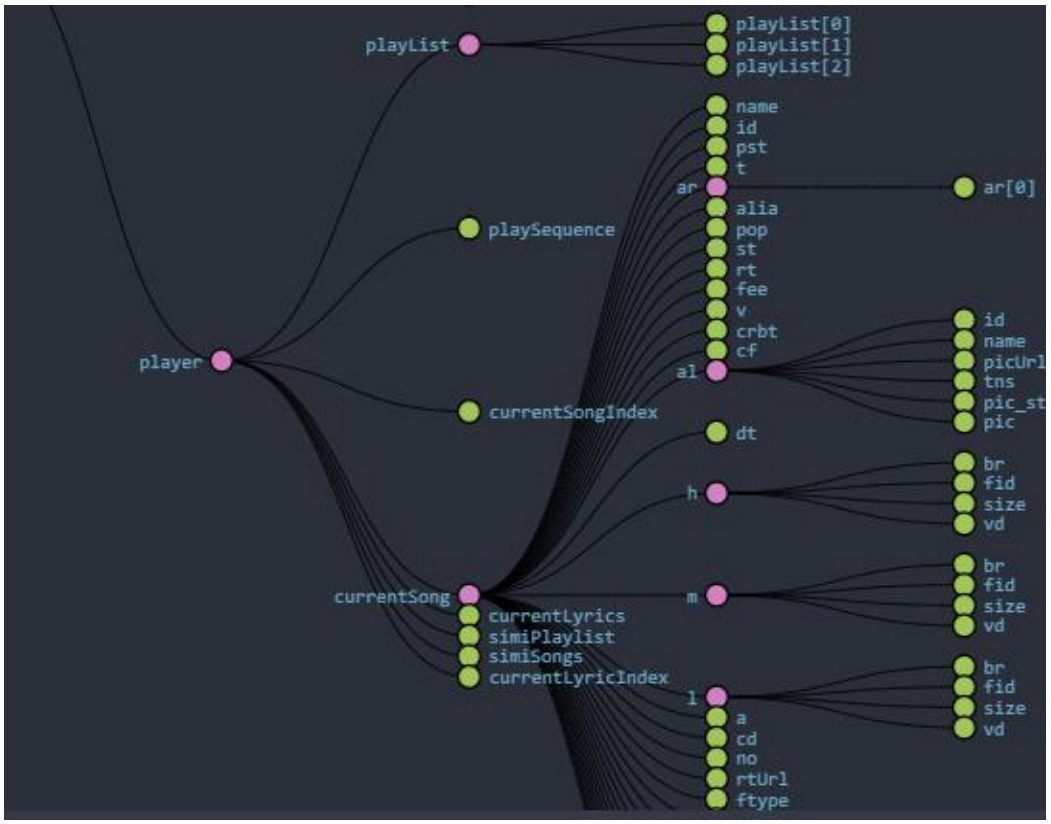


Рисунок 3.7 – Вигляд reducer player

Завдяки redux devtools можна побачити з чого складається отремий редюсер і які дані в ньому зберігаються.

Це дає змогу прослідкувати через який окремий action був змінений стор. А саме через прослідкувати весь ланюг зміни від action і його типу відправки чого через dispatch та обробра його в редюсері і зміна самого стора. На рис. 3.7

зображено редюсер який відповідає за програвача і можна прослідкувати в деревовидній структурі як реагує на це стор.

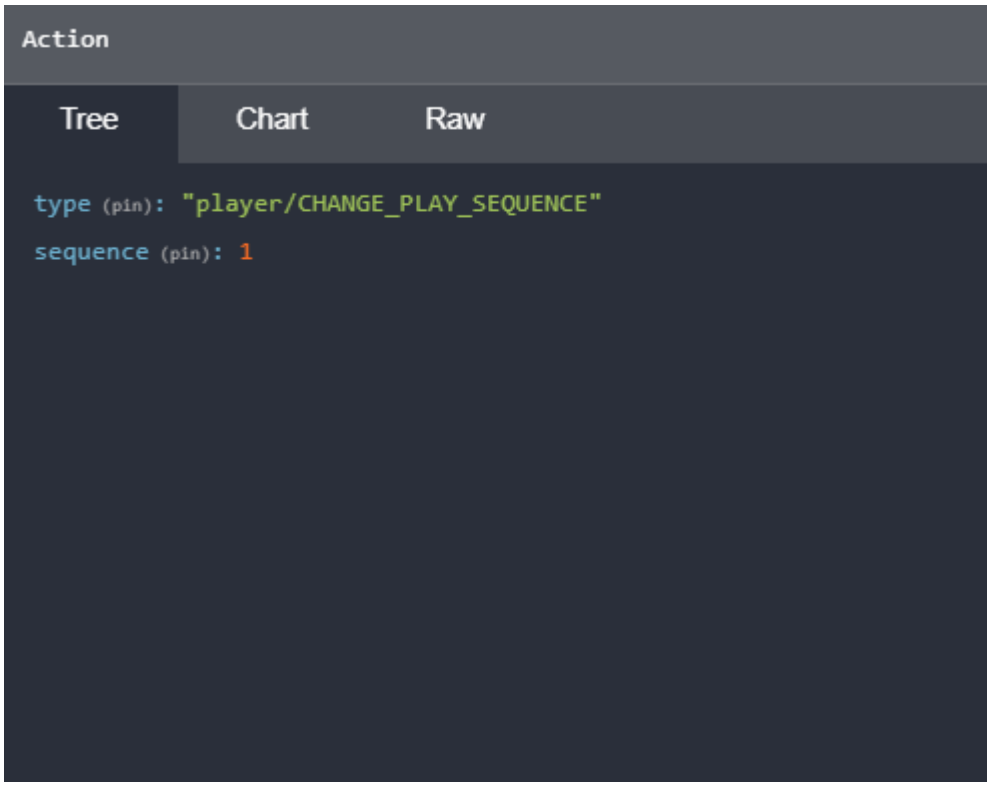


Рисунок 3.8 – Action

На даному Рисунок зображено який actionCreators змінює дані до стору.

Далі буде описано сторінка під назвою album

Таблиця 3.9 – Таблиця сторінки album.

function Album()	Компонент який є контейнером для інших компонентів сторінки для альбомів. Він розміщує в собі інші компоненти. Використовується HOC React.memo.
function TopAlbum()	Компонент який відображає категорію з альбомами аудіозаписів так званих “топові” або популярні. Даний компонент містить інші компоненти описані в розділі про основні компоненти(такі як пагінація). Інформація в компонент надходить із стору завдяки хуку useSelector. Містить логіку в зміни сторінки і

	логіку для обрання зі спику альбому. Всі ці дії виконує useDispatch. Інші складові візуальні в компоненті використовують styled-componets.
function HotAlbum()	Компонент який відображає категорію з альбомами аудіозаписів так званих “гарячі” або які входять топ чарти. Даний компонент містить інші компоненти описані в розділі про основні компоненти(такі як пагінація). Інформація в компонент надходить із стору завдяки хуку useSelector. Містить логіку для обрання зі спику альбому. Всі ці дії виконує useDispatch. Інші складові візуальні в компоненті використовують styled-componets.

В таблиці Таблиця 3.9 – Таблиця сторінки album, описано всі основні компоненти та логіку яку вони виконують.

Таблиця 3.10 – Таблиця actionCreators.

changeHotAlbumsAction	Змінити топ чартовий альбом
changeTopAlbumAction	Змінити популярні альбом
changeTopTotalAction	Змінити кількість альбомів

В «Таблиця 3.10» – Таблиця actionCreators описано за які дії відповідають окремі action

Рисунок 3.9 – reducer album

На рисунку зображено вигляд редюсера album.

Далі буде описано сторінка під назвою artict

Таблиця 3.11 – Таблиця сторінки artict.

function Artist()	Компонент який є контейнером для інших компонентів сторінки для відображення артистів. Він розміщує в собі інші компоненти. Використовується HOC React.memo.
function ArtistCategory()	Компонент який відображає артистів за категоріями. Інформація надходить зі стора за допомогою хука useSelector. На основі цих даних рендеряться різні категорії з виконавцями. При події кліку на вибраного артиста в категорії відбувається useDispatch який сповіщає про це стор. Інші елементи візуальні створені за допомогою styled-components.
function ArtistList()	Компонент який відповідає за відображення список виконавців.Компонент є контейнером для двох інших елементів. Дані надходять через хук useSelector.
function ArtistItemV1(props)	Компонент який відображає окремого виконавця. Дані отримує через пропси.
function AlphaList()	Компонент який відповідає за вибраний список і відображенням певного типу виконавців. Дані отримує зі стору через useSelector. А також слідкує за діїю кліку

	після чого викликає useDispatch в середину хуку useEffect.

В Таблиця 3.11 – Таблиця сторінки artict описано всю структуру і логіку компоненту artict.

А також зображено у древовдній структурі представлення редюсера artist на рис. 3.10

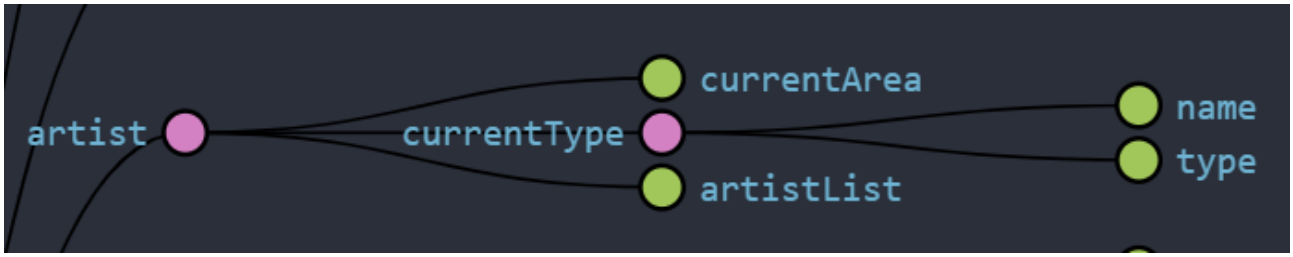


Рисунок 3.10 – reducer artist

Як можна спостеріга для даного редюсера є кілька різних action які призводять до певних змін.

Далі буде описано сторінка під назвою djradio

Таблиця 3.12 – Таблиця сторінки djradio.

function Djradio()	Компонент який є конейнером для інших компонентів сторінки для відображення радіо.
function RadioCategory()	Компонент який відповідає за категорії радіо. Інформація в компонент надходить через стор useSelector.
function RadioRanking()	Компонент який відображає рейтинг і також дозволяє переслинути сторінку з іншими рейтингами. Інформація в компонент надходить через стор useSelector.
function RadioRecommend()	Компонент який відображає

	рекомендовані. Інформація в компонент надходить через стор useSelector.

В Таблиця 3.12 – Таблиця сторінки djradio писано всю структуру і логіку компоненту djradio.

Таблиця 3.13 – Таблиця actionCreators.

changeCategoryAction	Змінити категорія аудіозаписів
changeCurrentIdAction	Змінити поточну аудіозапис
changeRecommendsAction	Змінити рекомендовані категорії
getRadioCategories	Отримати категорії

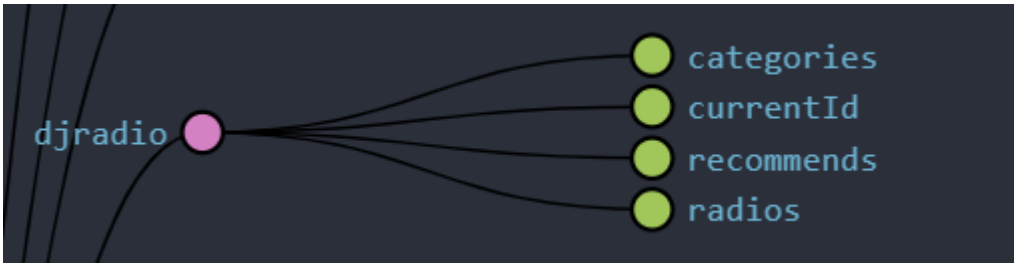


Рисунок 3.11 – reducer djradio

На рис. 3.11 зображено в деревовидній структурі редюсер artist

Далі буде описано сторінка під назвою djradio

Далі буде описано сторінка під назвою ranking

Таблиця 3.14 – Таблиця сторінки ranking.

function Ranking()	Компонент який є конейнером для інших компонентів сторінки для відображення рейтингу.
function RankingHeader()	Компонент який відповідає за

	категорії відображення інформації про рейтинг. Інформація в компонент надходить через стор завдяку useSelector.
function RankingList()	Компонент який відображає рейтинг з виконавцями та аудіозаписами. Інформація в компонент надходить через стор useSelector.
function TopRanking()	Компонент який відображає рейтинг з спеціальним рейтингом по замовчуванням . Інформація в компонент надходить через стор useSelector. При дії на зміну іншого рейтингу відправляє useDispacth, також в цьому компоненті логіка для добавлення власного “лайку”.

В Таблиця 3.14 – Таблиця сторінки ranking писано всю структуру і логіку компоненту ranking.

Таблиця 3.15 – Таблиця actionCreators.

changeTopListAction	Змінити вибірку плейлистів
changeCurrentIndex	Змінити поточний плейлист
getRanking	Отримати плей лист

В табл. 3.15 наведено всі action які створюються завдяки actionCreators.

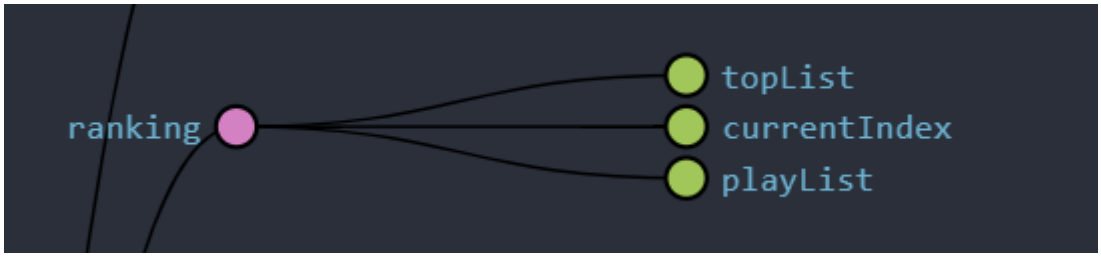


Рисунок 3.12 – reducer ranking

На рис. 3.12 зображено в деревовидній структурі редюсер runking. Та всі його action які певним чином оброблює цей редюсор.

Далі в табл. 3.16 буде описана сторінка ranking , яка в свою чергу відповідає за відображення популярних аудіозаписів.

Таблиця 3.16 – Таблиця сторінки ranking.

function Songs()	Компонент який є конейнером для інших компонентів сторінки для відображення аудіозаписів. Та виконує логіку для надання стору актальних даних при дії.
function SongsCategory()	Компонент який відповідає за відображення категорії аудіозапису. Інформація в компонент надходить через стор завдяку useSelector.
function SongsHeader()	Компонент який відображає інформацію про виконавця. Інформація в компонент надходить через стор useSelector.
function SongsList()	Компонент який відображає список аудіозаписів. Інформація в компонент надходить через стор useSelector. При дії на зміну іншого рейтингу відправляє useDispacth, також в цьому компоненті логіка для добавлення

	власного “лайку”.

В табл. 3.16 таблиця сторінки ranking писано всю структуру і логіку компоненту ranking. А далі в табл. 3.17 представлені генератори action.

Таблиця 3.17 – Таблиця actionCreators.

getCategory	Отримати всі категорії аудіозаписів
changeCurrentCategoryAction	Змінити поточну категорію
getSongList	Отримати категорії аудіозаписів

А на рис. 3.13 зображено в древовидній структурі це представлення редюсера song.

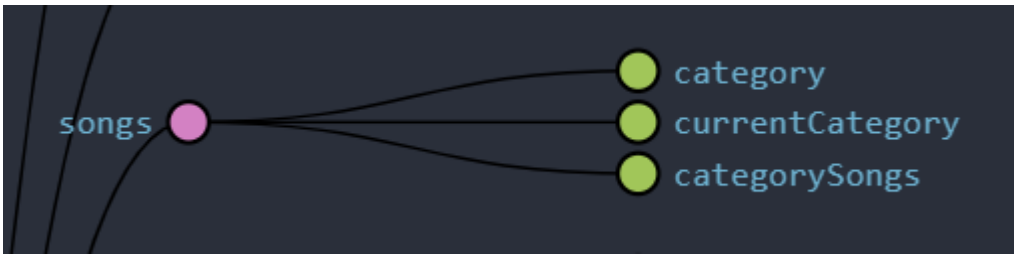


Рисунок 3.13 – reducer song

Далі буде описано сторінка під назвою ranking та наведені всі компоненти які є в даному модулі та містяться в табл. 3.18.

Таблиця 3.18 – Таблиця сторінки ranking.

function Recommend()	Компонент який є контейнером для інших компонентів сторінки для відображення рекомендованих аудіозаписів.
function HotRadio()	Компонент який відповідає за відображення категорії рекомендованого радіо. Інформація в

	компонент надходить через стор завдяку useSelector.
function HotRecommend()	Компонент який відображає інформацію розділу із найбільш популярними рекомендаціями
function NewAlbum(props)	Компонент який відображає рекомендовані альбоми. Інформація в компонент надходить через стор useSelector.
function RankingList()	Компонент відображає інший компонент і відображає всі списки з рекомендаціями. Дані надходять із стора через useSelector.
function SettleSinger()	Компонент відображає інформацію про виконавця та його аудіозапит. Дані отримує з стора через useSelector.
function TopBanner()	Компонент який відповідає за відображення компоненту з категорії рекомендоване. Також розміщенна логіка пеерелистування рекомендованих аудіозаписів. Дані надходять через useSelector.
function UserLogin()	Компонент який відповідає за перевірку чи авторизований користувач.

В табл. 3.18 ranking писано всю структуру і логіку компоненту ranking. Далі В табл. 3.19 буде представлено генератори action для даного редюсера.

Таблиця 3.19 – Таблиця actionCreators.

getBanner	Отримати дані
getRecommend	Отримати рекомендовані дані
getAlbum	Отримати рекомендовані альбоми
getTopData	Отримати дані по рекомендованих категоріях
changeNewListAction	Змінити на новий список рекомендованих
changeOriginListAction	Змінити свій список рекомендованих
getSettleSingers	Отримати окремо аудіозапис

На рис. 3.13 зображено в древовидній структурі редюсер recommend.

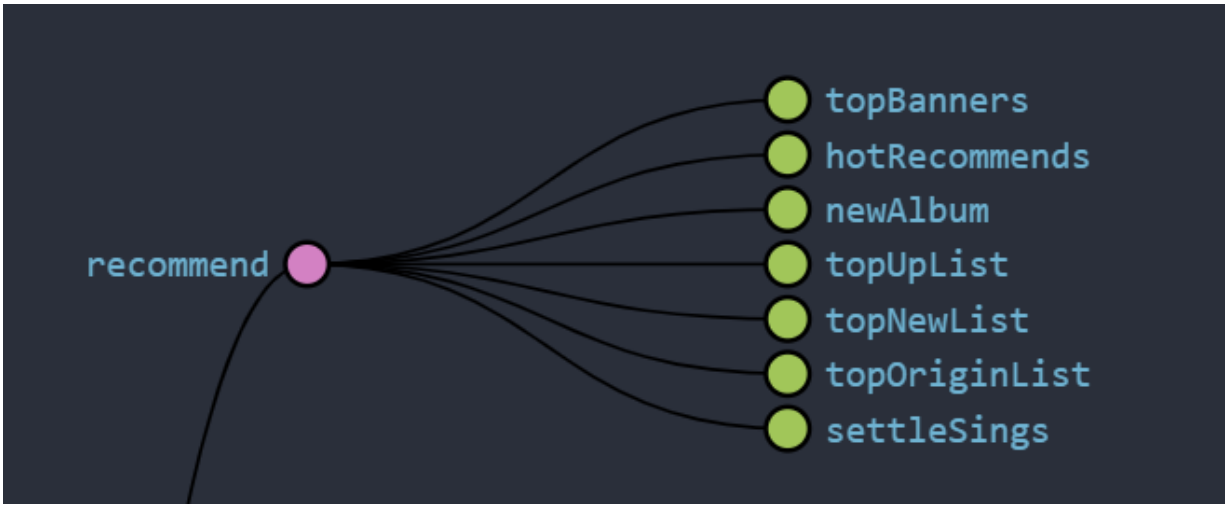


Рисунок 3.13 – reducer recommend

В даному розділі описано побудовану систему управлінням сервісами.

3.6 Вибір технологій для серверної частини

Почнемо опис з самого простого фреймворку, використовуваного на платформі Node.js.

Express використовується для розробки додатків досить давно і завдяки своїй стабільності міцно займає позицію одного з найпопулярніших фреймворків Node.js.

Для цього фреймворка існує велика кількість докладних інструкції та описів, які складені розробниками, перевірити його ефективність на практиці. Тому саме з Express рекомендується починати роботу, якщо ви маєте намір навчитися створювати додатки на платформі Node.js. Погодьтеся, набагато розумніше скористатися накопиченим і перевіреним досвідом, ніж заново винаходити велосипед.

Основна особливість цього фреймворка полягає в тому, що для Express характерний невеликий обсяг базового функціоналу. Всі інші необхідні вам функції потрібно буде добирати за рахунок зовнішніх модулів. По суті, Express в чистому вигляді - це сервер і у нього може не бути жодного модуля. Завдяки такому мінімалізму розробник спочатку отримує в своє розпорядження легкий і швидкий інструмент, який він може розширювати і розвивати. При цьому важливо, що вибір модулів для Express не пов'язаний ні з якими обмеженнями: ні з кількісними, ні з функціональними.

В результаті, цей фреймворк забезпечує розробнику можливість вирішувати будь-які завдання, не обмежуючи його при цьому у виборі засобів. З одного боку, не може не радувати той факт, що відсутність готових універсальних рішень фактично означає, що кожне створюване додаток буде унікальним.

З іншого боку, розробнику потрібно самостійно відбирати і організовувати модулі, а це передбачає великий обсяг роботи і відповідно, вимагає від розробника більше часу і зусиль.

Плюси:

- простота
- гнучкість
- хороша масштабованість
- розвинене співтовариство
- докладна документація
- широкий вибір модулів

Мінуси:

- великий обсяг ручної роботи
- використовується застарілий підхід callbacks функцій

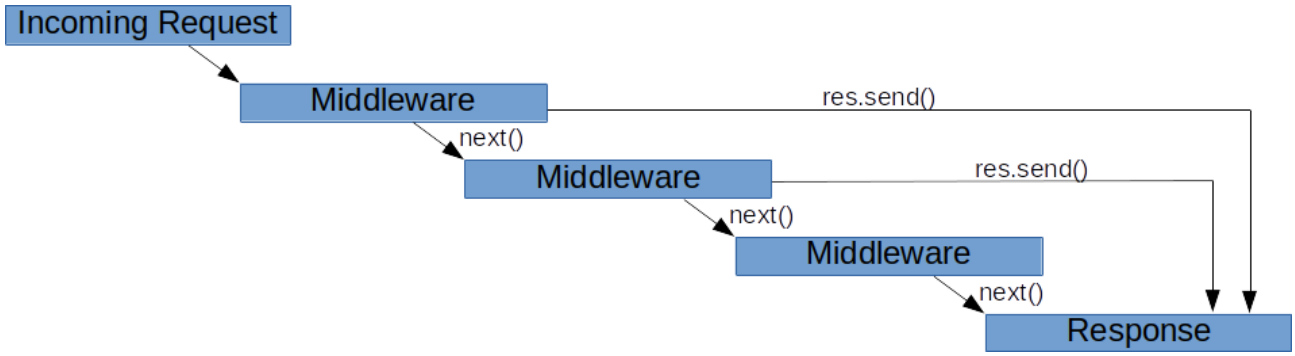


Рисунок 3.14 – Middleware в фреймворку

Функції тимчасової роботи (middleware) - це функції, які мають доступ до об'єкта запиту (req), об'єкту відповіді (res) і до наступної функції тимчасової роботи в циклі "запит-відповідь" додатка. Наступна функція тимчасової роботи, як правило, позначається змінної next.

Функції тимчасової роботи можуть виконувати такі завдання:

- Виконання будь-якого коду.
- Внесення змін до об'єкти запитів і відповідей.
- Завершення циклу "запит-відповідь".
- Виклик наступної функції тимчасової роботи з стека.

Якщо поточна функція тимчасової роботи завершує цикл "запит-відповідь", вона повинна викликати next () для передачі управління наступної функції тимчасової роботи. В іншому випадку запит зависне.

Додаток Express може використовувати такі типи проміжних оброблювачів:

- Проміжний обробник рівня додатки
- Проміжний обробник рівня маршрутизатора
- Проміжний обробник для обробки помилок
- Вбудовані проміжні обробники
- Проміжні обробники сторонніх постачальників ПО

Проміжні обробники рівня додатки і рівня маршрутизатора можна завантажувати за допомогою необов'язкового шляху для монтування. Також можна завантажити послідовність функцій тимчасової роботи одночасно, в результаті чого створюється допоміжний стек системи проміжних оброблювачів в точці монтування.

Passport.js - це гнучке проміжне програмне забезпечення для автентифікації (що дозволяє користувачам входити в систему), яке можна повністю налаштувати і чудово працює з connect / express.

Він гнучкий в тому сенсі, що дозволяє застосовувати різні стратегії автентифікації [13] (думайте через Twitter, через власну базу даних користувачів - встановлюється за допомогою окремих модулів, які можна поєднувати), і дозволяє нам вказати будь-який маршрут або вихід під час автентифікації.

Місцева стратегія дозволяє нам автентифікувати користувачів, шукаючи їх дані в базі даних програми. У ньому є кілька чудових прикладів того, як ним користуватися.

У цьому дописі ми проходимо потік автентифікації, наступний допис обговорює деякі конкретні випадки використання за допомогою паспорта.

Існує три основні частини використання паспорта.js:

- Потрібен модуль та використання його проміжного програмного забезпечення express.initialize () та.

- Налаштування паспорта принаймні з однією стратегією та налаштування методів serializeUser паспорта та deserializeUser.
- Визначення маршруту, який використовує паспорт. Автентифікація проміжного програмного забезпечення для фактичної автентифікації користувача.

Nodemon - це утиліта командної строки.Вона вкладає в оболочку ваше додаток Node, спостерігає за файловою системою та автоматичним перезапуском процесу.

Що таке MongoDB.

MongoDB - це база даних, яка зберігає ваші дані у документах. Як правило, ці документи мають JSON (* JavaScript Object Notation - текстовий формат обміну данними, заснований на JavaScript[20].

Одна з основних особливостей MongoDB - гнучкість структури даних. Крім того властивості можуть бути відсутні в інших документах користувачів колекцій користувачів. Це вигідно відрізняє MongoDB від бази даних SQL (* мова структурованих запитів - мова структурованих запитів), наприклад, MySQL або Microsoft SQL Server, в яких для кожного об'єкта, що зберігаються в базі даних, необхідна фіксована схема.

За рахунок можливостей створення динамічних об'єктів, які зберігаються у відеодокументах в базі даних, у іграх, що вступають до Mongoose.

Mongoose - це ODM (* Map Document Mapper - об'єктно-документний відображувач). Це означає, що Mongoose дозволяє вам визначити об'єкти строго типової схеми, що відповідає документу MongoDB[20].

Мінуси MongoDB

Потенційні переваги MongoDB були величезні, особливо для певних класів проблем. Якщо прочитати вищенаведений список без розуміння контексту і не маючи досвіду, то може скластися враження, що MongoDB дійсно революційна СУБД. Єдина проблема полягала в тому, що перераховані вище переваги супроводжувалися низкою застережень, деякі з яких вказані нижче.

Втрата транзакцій: транзакції є основною особливістю багатьох реляційних баз даних (не всіх, але більшості). Транзакційність означає, що ви можете виконувати декілька операцій атомарному і можете гарантувати, що дані залишаться узгодженими. Звичайно, з базою даних NoSQL транзакційність може бути в рамках одного документа або ви можете використовувати двофазні коммітов, щоб отримати трансакціонної семантику. Але вам доведеться самому реалізувати цю функціональність ... що може бути складною і трудомісткою завданням. Часто ви не усвідомлює проблеми, поки не побачите, що дані в БД потрапляють в неприпустимі стану, тому що неможливо гарантувати атомарность операцій. Примітка: багато мені повідомили, що в минулому році в MongoDB 4.0 з'явилися транзакції, але з низкою обмежень. Висновок зі статті залишається тим самим: оцініте, наскільки технологія відповідає вашим потребам.

Втрата реляційної цілісності (зовнішні ключі): якщо в ваших дані є відносини, то вам доведеться застосовувати їх в додатку. Наявність БД з дотриманням цих відносин зніме значну частину роботи з програми, але, отже, з ваших програмістів.

Відсутність можливості застосовувати структуру даних: строгі схеми іноді стають великою проблемою, але це також і потужний механізм хорошого структурування даних, якщо грамотно їх використовувати. Документні БД, такі як MongoDB, забезпечують неймовірну гнучкість схеми, але ця гнучкість знімає відповідальність за збереження даних в чистоті. Якщо не подбати про них, то в кінцевому підсумку доведеться писати в додатку багато коду для обліку даних, які зберігаються не в тій формі, яку ви очікуєте.MongoDB підтримує перевірку схеми: вона корисна, але не надає ті ж гарантії, що в реляційної БД. Перш за все, додавання або зміна перевірки схеми не впливає на існуючі дані в колекції. Ви самі повинні переконатися, що оновлюєте дані відповідно до нової схеми. Вирішуйте самі, чи достатньо цього для ваших потреб.

В даному розділі було зроблено опис інструментів які використовувались для створення серверної частини.

3.7 Тестування системи

Основною метою створення системи для надання сервісів є вдосконалення швидкодії системи. Це потрібно для зменшення візуальних затримок для користувача і правильної роботи системи.

При розробці односторінового застосунку було примінено та описано архітектурний патерн Flux. Завдяки однонаправленому потоці досягнути максимальна швидкодія.

Також за допомогою правильно використаного фреймворку та всіх його можливостей досягненна максимальна швидкодія.

Таблиця 3.20 – Результати примінення React.lazy.

Event	WIFI	4G	Fast 3G
FP	658ms (+6%)	791ms (-3%)	2258ms (-5%)
FMP	2135ms (+2%)	2286ms	3725ms (-3%)

Використання тільки компонента lazy при першому запуску додатка та результати наведені в табл. 3.20.

Після використання тільки одного інтрументу lazyне було досягнути очікуваної швидкодії. В табл 3.21 наведено приклад виділення core компонента і тестування швидкодії.

Таблиця 3.21 – Результати примінення React.lazy через LazyComponent.

Event	WIFI	4G	Fast 3G
FP	487ms (-21%)	570ms (-30%)	1829ms (-23%)
FMP	1965ms (-6%)	2048ms (-10%)	3468ms (-9%)

В абсолютних числах звучить, звичайно, не так приємно, але 400ms на fast3g ми все-таки виграли.

Після цих дослідженню стає зрозуміло, що стандартні інструменти які має фрейворк не дозволяють отримати потрібну швидкість для системи. Для цього

було зроблено удосконало глобальний систему управлінням станом додатку в саме Reselect, який в свою чергу робить мемоізації для звернення в глобальний store. Отримані нові дані наведені у табл. 3.21.

Таблиця 3.22 – Результати примінення React.lazy через LazyComponent.

Event	WIFI	4G	Fast 3G
FP	136ms (-44%)	201ms (-8%)	666ms (-15%)
FMP	1519ms (-12%)	1418ms (-18%)	3743ms (+8%)

А на рис. 3 15 показано результати кінцевого тестування розробленої системи.

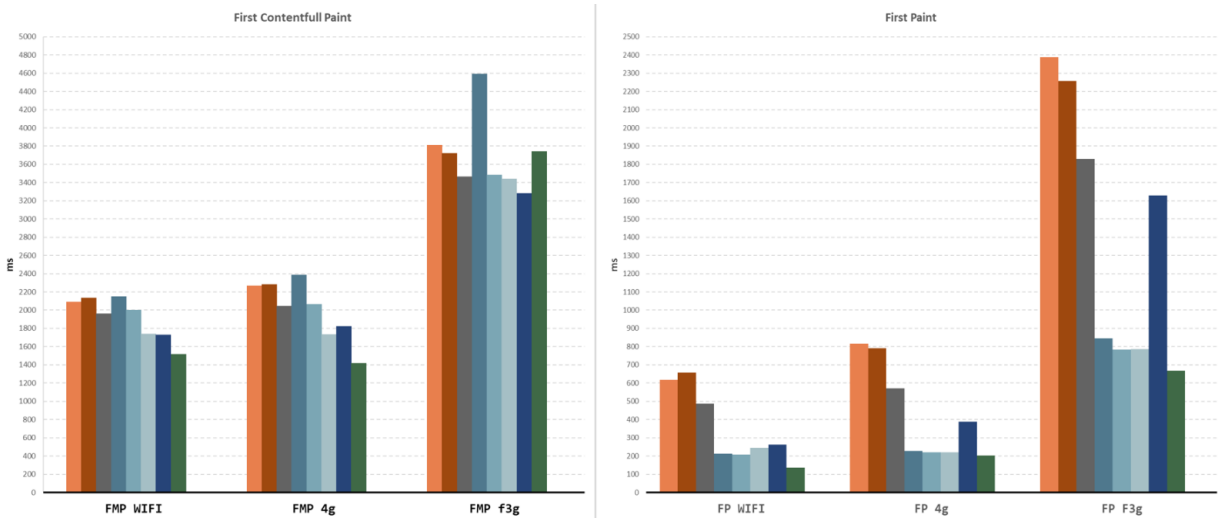


Рисунок 3.15 – Результати тестів швидкості роботи

Сам по собі, React - не швидкий, але він пропонує всі інструменти, щоб зробити швидким додаток будь-якого розміру.

Це виглядає нелогічним, особливо, коли безліч фреймворків пропонують альтернативи React, стверджуючи, що вони швидше його в N раз. Але React ставить на перше місце зручність і досвід розробника, а не продуктивність. Це та причина, по якій розробка великих додатків з React це приємний досвід, без поганих сюрпризів і зі стабільним темпом реалізації.

Тому використання тільки базових інтструментів не досить для покращення швидкості роботи.А на рис. 3.16 зображено використання ресурсів при повній оптимізації.

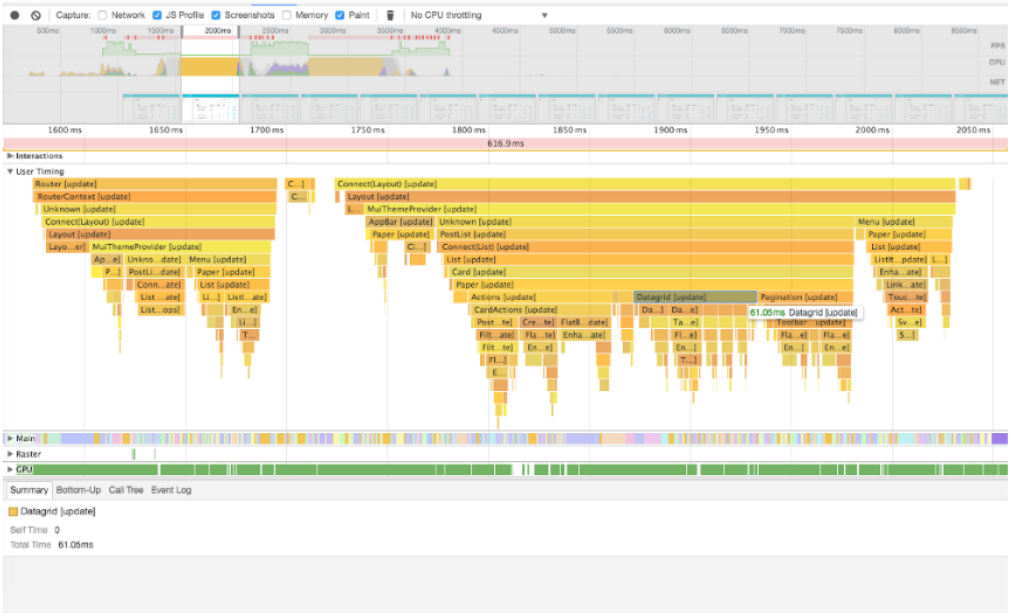


Рисунок 3.16 – Результати тестів швидкості роботи DevTools

Тому для створення системи просто на базі фреймворку React яка б задовільняла певні характеристики не достатньою просто використовувати інструмент.

Потрібно дослідити роботу інструменту, як працює система оптимізації та архітектурні паттерни які використовує даний фреймворк.

3.8 Висновки

В даному розділі сформовано опис об’єкта дослідженні, а саме реалізація односторінкового застосунку так званого SPA. Для архітектурного основного паттерна вибрано Flux, який є схожим до стандартних MV* паттернів, які зазвичай мають примінення у даних системах.

Flux паттерн який реалізований у Redux використаний для одностороннього потоку даних, який забезпечує системі надання сервісів у музичній галузі максимально просту і швидку роботу. Яку легко розширювати та підтримувати.

Реалізація системи здійснювалась на фрейворку React який виступає в ролі відображення елементів та займається рендерингом, маршрутизацією. Всі дані в систему надходять з системи управління станом Redux

Для отримання даних була реалізована серверна частина. Яка надає дані в глобальну систему управління станом.

А в останньому розділі було протестовано реалізована система на відповідність виконання мети дисертації. Всі критерії які були модельовані та поставлені виконано.

4.ІНСТРУКЦІЯ ПО КОРИСТУВАННЮ

4.1 Область застосування

Інструкція призначена для надання необхідної інформації щодо користування програмним забезпеченням (надалі – ПЗ), а також – для надання опису можливих помилок при використанні ПЗ.

ПЗ встановлюється на ПК і виконує наступні функції:

Реєстрація користувачів

Аутентифікація та авторизація користувачів

взаємодія з користувачами (виводиться список рекомендованих аудіозаписів, перелік різних налаштувань для вибору необхідних аудіозаписів користувачу)

взаємодія з користувачем через навігаційнц панель

4.2 Головна сторінка

Основна частина компонентів та перша сторінка яка відображається при запуску додатка.

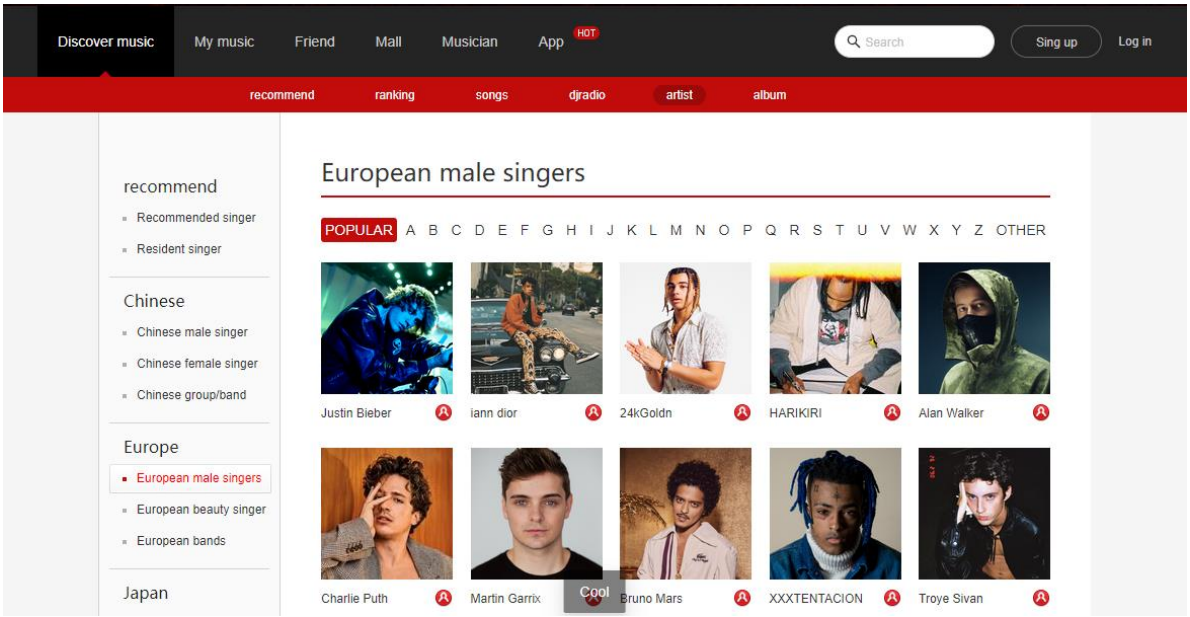


Рисунок 4.1 – Графічний вигляд головної сторінки

На рис. 4.1 зображено головну сторінку при запуску додатку. На зображенні розміщенно навігаційну панель

4.3 Авторизація

На рис. 4.2 зоображено інтерфейс для логіну в систему.

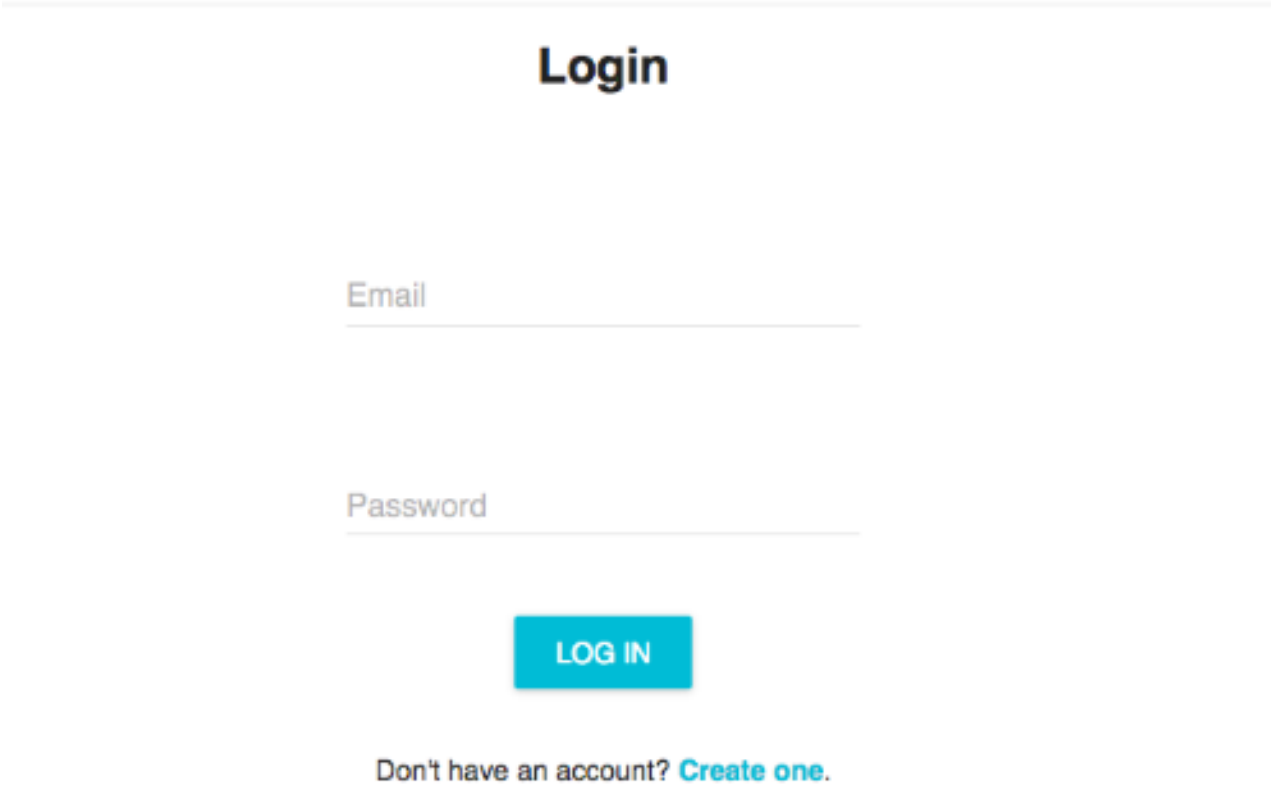


Рисунок 4.2 – Графічний вигляд форми входу

На рис. 4.2 зоображено інтерфейс для логіну в систему.

Для того щоб авторизувати користувача необхідно з головної сторінки в навігаційному меню натиснути кнопку інтерфейсу Log in. При натисненній цієї кнопки користувач буде бачити перед собою сторінку логіна.

Для авторизації в системі користувач повинен ввести свою електронну пошту а також пароль який він вказував в реєстрації в системі.

Коли дані будуть введені користувачем, перед відправлення інформації на сервер(творення запиту) інформація введена користувачем спочатку має пройти перевірку на стороні веб додатку. Для цього в додатку налаштована система валідації.

Якщо користувач вводить невалідну інформацію для системи (наприклад шаблоном для поля password є 8 значне число). І натисне кнопку LOG IN, то перед відправленням цієї інформації на сервер веб додаток перехватить контроль і повідомить користувача про невалідність даних вказаних ним. І вкаже на поле ввода де була зроблена помилка.

Якщо ввалідність даних підходить для клієнтської сторони то відбувається запит на сервер. Сервер має авторизувати користуча, або повернути інформацію про те що такого користувача не істує чи введені невірні дані

4.4 Реєстрація

На рис. 4.3 зображено форму реєстрації. Яку буде бачити користувач.

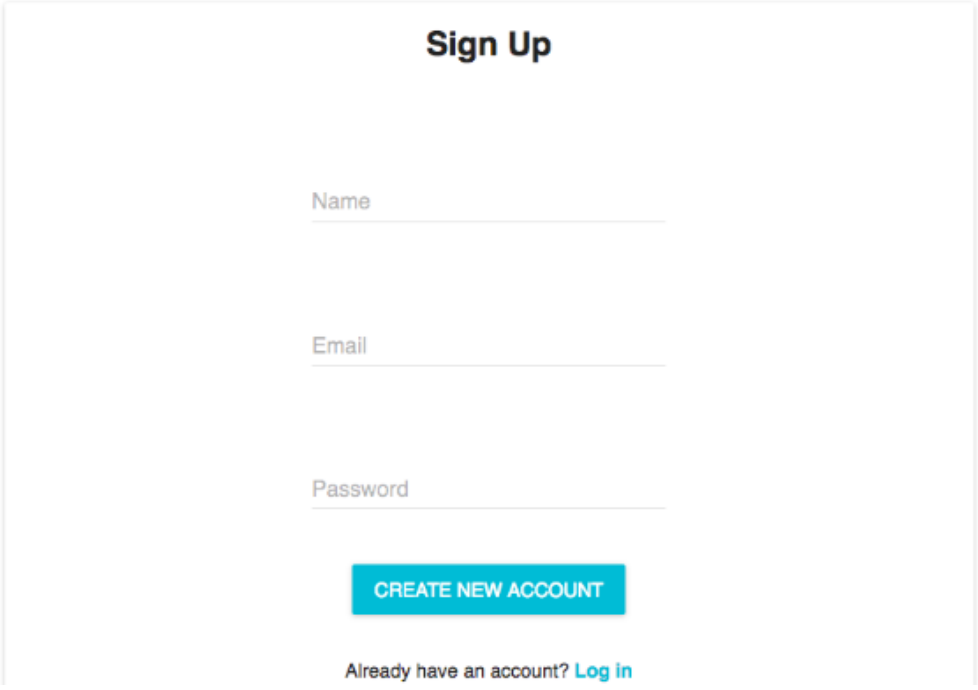


Рисунок 4.3 – Графічний вигляд форми реєстрації

Для того щоб зареєструвати користувача необхідно з головної сторінки в навігаційному меню натиснути кнопку інтерфейсу Sing in. При натисненній цієє кнопки користувач буде бачити перед собою сторінку реєстрації.

Для реєстрації в системі користувач повинен ввести свою електронну пошту, створити пароль та нікнейм та натиснути кнопку CREATE NEW ACCOUNT для реєстрації в системі.

Коли дані будуть введені користувачем, перед відправлення інформації на сервер(творення запиту) інформація введена користувачем спочатку має пройти перевірку на стороні веб додатку. Для цього в додатку налаштована система валідації.

Якщо користувач вводить невалідну інформацію для системи (наприклад шаблоном для поля password є 8 значне число). І натисне кнопку CREATE NEW ACCOUNT, то перед відправленням цієї інформації на сервер веб додаток перехватить контроль і повідомить користувача про невалідність даних вказаних ним. І вкаже на поле ввода де була зроблена помилка.

Якщо ввалідність даних підходить для клієнтської сторони то відбувається запит на сервер. Сервер має зареєструвати користуча, або повернути інформацію про те що такий користувач уже.

4.5 Основні частини

Користувачеві дозволяється знаходження в системі не авторизованому, але неавторизований користувач має певні обмеження в користуванні додатку.

При виборі регіону відображається вибрані виконавці. Користувач може сортувати отриманих виконавцем за алфавітним порядком їх імен, або завдяки створенним унікальним фільтром POPULAR.

На рис. 4.4 зображено всі можливі фільтри для вибору по музикантам для знаходження потрібного або музиканта або його аудіозапису.

Також існує фільтри для пошуку альбомів.

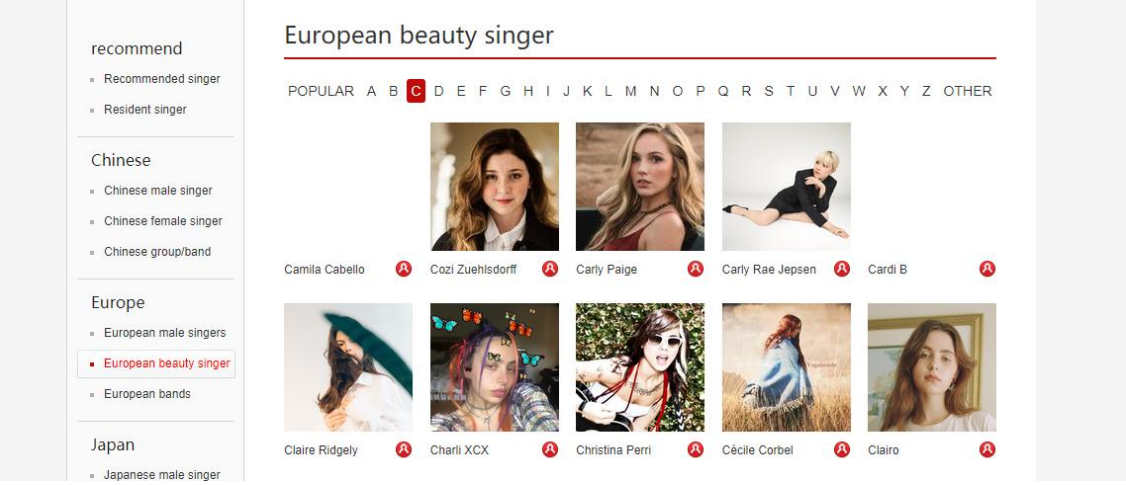


Рисунок 4.4 – Графічний форми фільтрування

Для вибору користувачеві певного виконавця створена бокова панель, яка відображає інформацію по регіонам. Щоб користувач міг зручно вибрати необхідний йому регіон.

4.6 Програвач

Рисунок 4.5 – Графічний форми програвача

Зображено на рис. 4.5 міні версія програвача, якою може користуватися юзер. Він має такі функції:

- Включити/Зупинити
- Далі
- Назад
- Пролиснути поточий аудіозапис
- Відкрити повну версію програвача
- Регулювати звук
- Зміна порядку програвання аудіозаписів
- Весь плей лист

Користувач може використовувати всі ці функції програвача та змогу сховати даного програвача. А також має змогу відкрити повну версію на рис. 4.6.

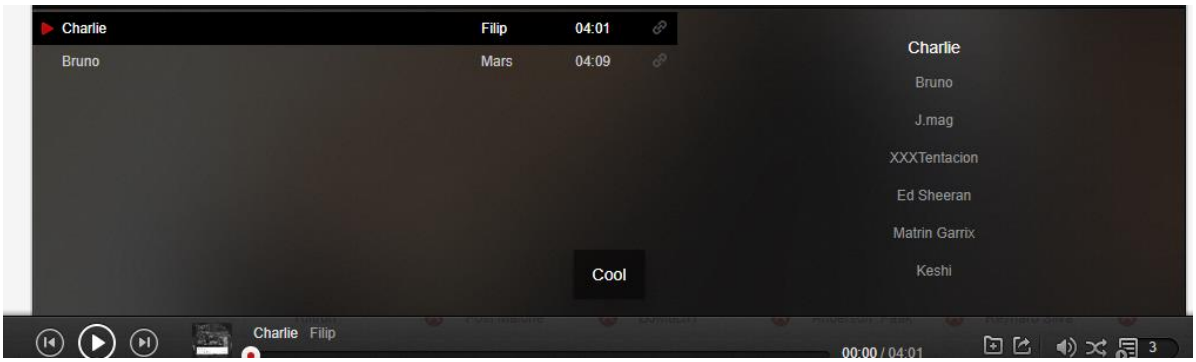


Рисунок 4.6 – Графічний форми повна версія програвача

При натисненні користувачем в область плейера він переходить в повний режим в якому крім озвучених функцій, отримує кілька нових.

В такому режимі користувач може побачити весь список вибраних ним аудіозаписів. Та список аудіозаписів яка система автоматично додає до плей листа якщо в ньому закінчились обрані користувачем аудіозаписи.

4.7 Рейтингова сторінка

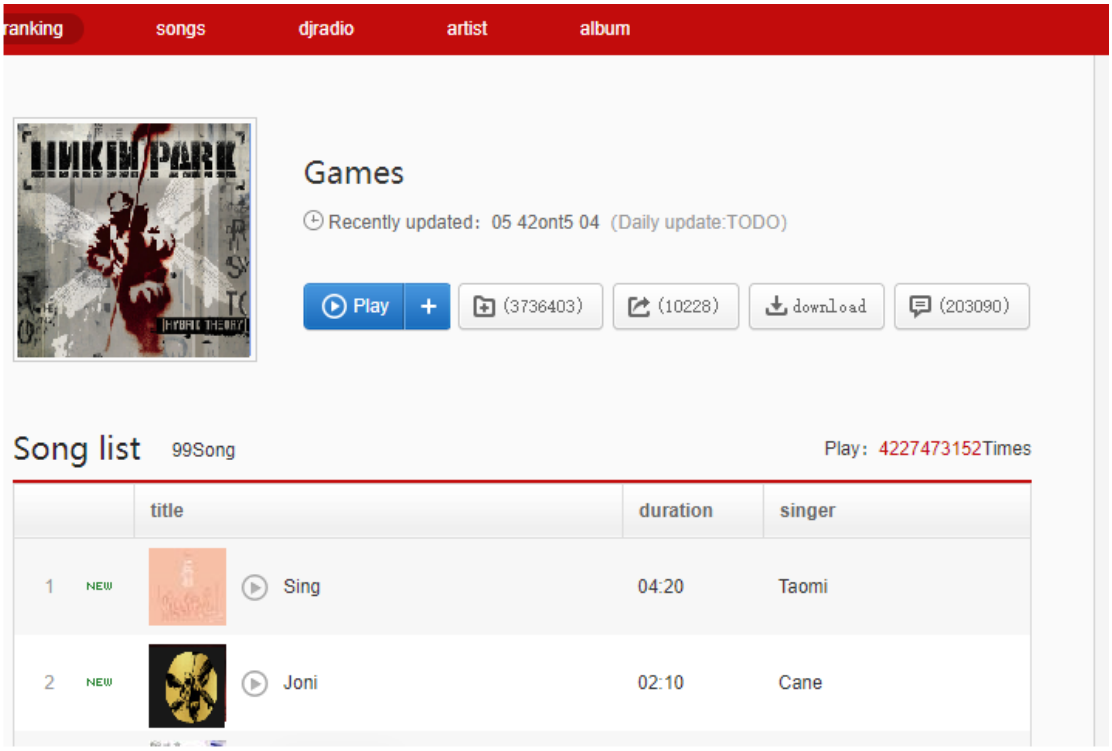


Рисунок 4.7 – Графічний вигляд оформлення альбому

На рис. 4.7 можна потрапити з головної сторінки, через навігаційну систему. Для переходу на дану сторінку потрібно натиснути кнопку ranking на головному меню

При натисненні кпоки користувач буде бачити перед собою найбільш популярніші альбоми, та має змогу передивитися весь альбом. Альбом має вигляд змішаних найпопулярніших аудіозаписів різних виконавців.

При пролистуванні сторінки користувач буде бачити весь список альбому та аудіозаписів які входять в даний альбом.

Якщо користувач має бажання прослухати композиція він може натиснути на кнопку яка розміщенна біля аудіозапису в альбомі і вона автоматично додається до програвача.

Що можу зробити користувач з альбомом:

- Також користувач має зможу завантажити весь альбом
- Додати альбом в свій плей лист
- Поділитися цим альбомом в соц мережах
- І написання коментаря та перегляд інших коментаріїв

4.8 Висновки

В розділі надано розгорнуті інструкції по користуванній розробленій програмній системі надання сервісів у музичній галузі. Описано та у графічній формі відображенно основні компоненти та елементи односторінкового застосунку. Графічний інтерфейс був розробленний завдяки аналізу схожих продуктів та вироблений на цьому аналізу власний, який позбавлений критичних присутніх у інших продуктах недостатків.

Було реалізовано у власному інтерфейсу основні властивості:

- Природність інтерфейсу
- Узгодженість інтерфейсу
- Узгодженість в межах продукту
- Узгодженість в межах робочого середовища
- Дружелюбність інтерфейсу (принцип «пробачення» користувача)
- Принцип «зворотного зв'язку»
- Простота інтерфейсу
- Гнучкість інтерфейсу

Властивості було спроектованно у 2 розділі і вони являлися предметом дослідження.

5 РОЗРОБКА СТАРТАП ПРОЕКТУ

5.1 Інформаційна карта проекту

У таблиці 5.1 наведено інформаційну карту стартап проекту.

Таблиця 5.1 – Інформаційна карта проекту

1. Назва проекту	MusicTime (Додаток для прослуховування музики)
2. Автори проекту	Мирослав Поліщук
3. Коротка анотація *(не більше 1/3 сторінки)*	Створення та реалізація проекту MusicTime. Орієнтується на людей, які полюбляють активний відпочинок. Можливість авторизації в додатку, встановлення власного плей листу,, приєднання до інших людей, моніторинг активностей, переписка з іншими людьми.
4. Термін реалізації проекту	13 місяців *Тривалість проекту (в місяцях)*
5. Необхідні ресурси	Інтелектуальні.Команда людей для розробки продукту програміст який буде реалізовувати додаток. І команда з (3 людей) для моніторингу і розповсюдженні проекта. Матеріальні. Необхідні матеріальні кошти для устаткування, оренди офіса, виплати заробітної плати і реклами.

	Фінансові.Устаткування (60 000 грн) Заробна плата(200 000 грн) Оренда офісу (156 000 грн), Реклама(60 000 грн).
	Перелік усіх необхідних ресурсів (фінансових, матеріальних інтелектуальниї та ін.)
6. Опис проблеми, яку вирішує проект	Основна проблема, що люди які прослуховують музику не мають можливості кооперації ,відстеження, створення нових маршрутів. Мій проект дає змогу відстежувати інших людей, кооперуватися з ними, створенні власних маршрутів, моніторинг власних маршрутів.

7. Головні цілі та завдання проекту	Завдання: 1.Створення зручного та інформаційного інтерфейсу , який буде використовуватися користувачами. 2.Подальший розвиток та удосконалення проекту. 3.Можливість збору фідбеку, для інтеграції нових можливостей. Цілі: Реалізація програмного продукту .

8. Очікувані результати *(Описати позитивні зміни, які відбудуться в результаті реалізації*

проекту після його завершення та в довгостроковій перспективі)
В результаті реалізації проекту він має всі шанси, щоб завойювати ринок, бути активно використовуваним, популярним. Проект має перспективу подальшого розвитку, за рахунок фідбеку від користувачів і досить гнучким, так як може задовільняти потреби майже кожного користувача.

5.2 Розподіл стартап проекту між учасниками

У таблиці 5.2 наведено розподіл стартап проекту між учасниками.

Таблиця 5.2 – Розподіл стартап проекту між учасниками

	Команда	**Проекти**	**Що потрібно зробити для проекту**	**Що потрібно зробити члену команди**
1	Олеся (Менеджер)	Проект Олесі	Задача менеджера Задача маркетинга Задача IT	Задача менеджменту для Олесі Задача менеджменту для Максима Задача менеджменту для Міші
2	Максим(Маркетолог)	Проект Максима	Задача менеджера Задача маркетинга Задача IT	Задача маркетингу для Олесі Задача маркетингу для Максима Задача маркетингу для Міші Задача маркетингу для Віктора
3	Міша(IT спеціаліст)	Проект	Задача менеджера	Задачу IT для

		Міши	Задача маркетинга Задача IT	Олесі Задачу IT для Максиму Задачу IT для Міші

В даній ситуації ми розглядаємо IT-стартап, заснований трьома випускниками ВНЗ.

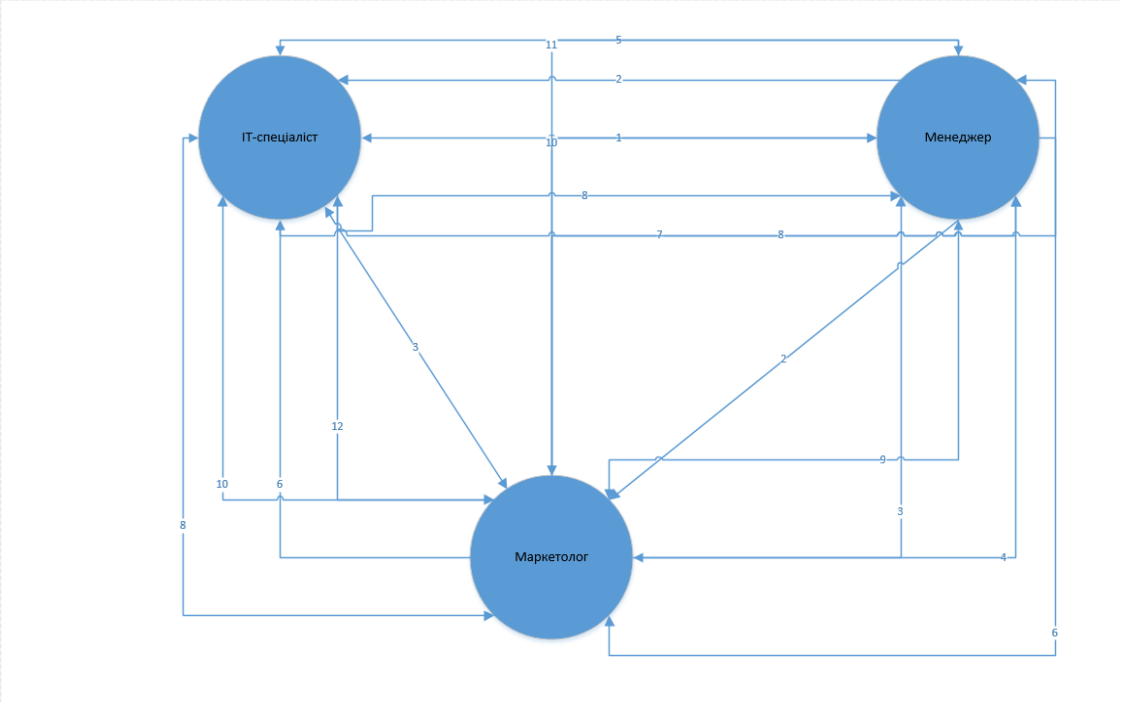
1. Генератор ідей, експерт з технологій.
2. Відповідальний за бізнес-складову.
3. Спеціаліст з технологій, аналітик.

КРОК 2

Таблиця 5.3 – Розподіл завдання між учасниками

	Необхідне завдання	К-кість місяців
1	Дослідження методів реалізації	1
2	Видача ТЗ	1
3	Розподіл обов’язків	0,5
4	Створення дизайну	2
5	Розробка ПЗ	2
6	Тестування ПЗ	1,5
7	Розробка конструкції	0,5
8	Оцінка і налагодження	2
9	Пошук інвесторів	0.5
10	Юридичне забезпечення	0.5
11	Рекламна кампанія	1
12	Аналітика	0.5

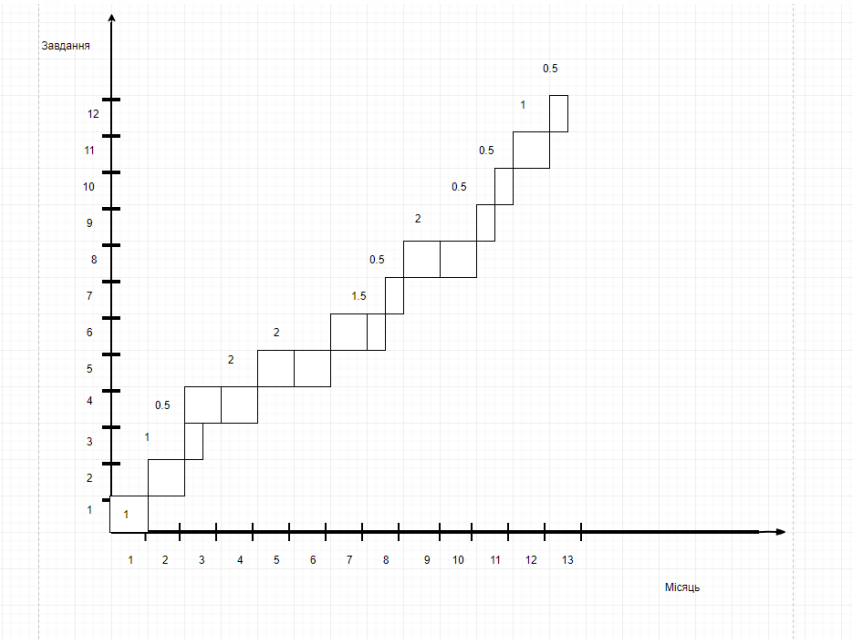
КРОК 3



1 МЕНЕДЖЕР = Вх/Вих = 12/13 = 0.92

2 МАРКЕТОЛОГ = Вх/Вих = 8/9 = 0.888

3 РОЗРОБНИК = Вх/Вих = 8/11 = 0.72



КРОК 4

1 МЕНЕДЖЕР = 0,8 + 0.33 + 0,2 + 1 + 0,5 + 0,5 + 0,2 + 1 + 0,2 + 0.2 + 0.3 + 0.7 = 5.93

2 МАРКЕТОЛОГ = 0,33 + 0.1 + 0.5 + 1 + 0,5 + 0,3 + 0.5+ 0.2 + 0.2 + 0.5 + 0.2 = 4.13

3 РОЗРОБНИК = 0.33 + 0.2 + 1 + 0.5 + 0,5 + 0.1 + 0.1 + 0.1 + 0.1 = 2.81

1 МЕНЕДЖЕР = 5.93/13 = 0,45

2 МАРКЕТОЛОГ = 4.13 / 13 = 0,31

3 РОЗРОБНИК = 2.81 / 13 = 0,21

Завантаженість МЕНЕДЖЕРА = 0.45/0,92 = 0,49

Завантаженість МАРКЕТОЛОГА = 0.31 / 0,888 = 0.34

Завантаженість РОЗРОБНИКА = 0.21 / 0,72 = 0.3

5.3 Визначення важливості факторів

Визначення важливості факторів щодо їх вкладу у створення та реалізацію стартапу

Фактор	**Вага (важливість)**
Ідея	7
Підготовка бізнес плану	8
Компетентність	9
Залученість і ризики	6
Обов'язки	8

Оцінювання важливості кожного фактора і внеску кожного учасника

Фактор	**Вага**	**Партнер 1**	**Партнер 2**	**Партнер 3**
Ідея	7	8	6	3
Підготовка бізнес плану	8	7	4	2
Компетентність	9	6	4	4
Залученість і	6	7	5	6

ризики				
Обов'язки	8	9	7	4

Визначення дольової участі у стартап проекті кожного учасника

Фактор	**Партнер 1**	**Партнер 2**	**Партнер 3**	
Ідея	3	6	8	
Підготовка бізнес плану	2	4	7	
Компетентність	5	4	6	
Залученість і ризики	6	5	7	
Обов'язки	4	7	9	
Разом	19	26	37	82
Процент	23.1	31.7%	45.1	100%

Визначення основних функцій

- Створення плей листу для користувача
- Можливість зберігати плей листи
- Підбір можливих плей листів
- Можливість відображення інших учасників
- Можливість кооперації з іншими людьми

Таблиця 5.4 .Визначення базової стратегії розвитку

№ п/п	*Обрана альтернатива розвитку проекту*	*Стратегія охоплення ринку*	*Ключові конкурентоспроможні позиції відповідно до обраної*	*Базова стратегія розвитку**

			альтернативи	
	Орієнтація на український ринок з можливостями виходу на міжнародний рівень	Робити ставку на зручність простоту, та інтелектуальні можливості розробленої системи.	Оптимальна кількість можливостей і зручність використання.	Стратегія дифиренційоваго маркетингу - користувач обирає всі функціїї під себе.

Таблиця 5.6 Визначення базової стратегії конкурентної поведінки

№ п/п	*Чи є проект «першопрохідцем» на ринку?*	*Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?*	*Чи буде компанія копіювати основні характеристики товару конкурента, і які?*	*Стратегія конкурентної поведінки**
	Розроблювана продукція є першопрохідцем на ринку	Компанія буде збирати та шукати нових споживачів. При виникненні - 35% клієнтів забирати у конкурентів, 65% нових.	Ні	Позиційна оборона Стратегія заняття конкурентної ніші

5.4 Висновки

Проектом є система для надання сервісів у музичній галузі.

Було розроблено і затверджено інформаційну карту стартап проекту. Проаналізовано необхідні ресурси, які потрібні для реалізації проекту. Розпланована реалізацію проекту, а також за його підсумками розроблено і затверджено чіткий план реалізації проекту, прив'язаний до календарного плану.

Рентабельність даної системи знаходиться на досить високому рівні, але надалі необхідно розвивати продукт як в плані технологій, так і маркетингу. Це пов’язано з тим, що розробка проводиться у сфері, де технології швидко застарівають та заміщуються, постійно постають все нові вимоги клієнтів.

На основі результатів проведеного аналізу можна зробити висновок, що подальший розробка даного продукту є доцільною, оскільки в галузі є потреба в рішеннях такого роду, а обране середовище є перспективним для розвитку страртап-проекту.

ВИСНОВКИ

У ході виконання магістерської дисертації було розроблено систему надання сервісів у музичній галузі.

Представлене програмне забезпечення та реалізація його архітектурних принципів справедлива для більшості платформ. Практична цінність та можливості реалізованого продукту було протестовано за допомогою тестів і представленно результати у графічних матеріалах.

Відповідне програмне рішення складається з наступних модулів:

— модуль для побудови середовища взаємодії між компонентами системи та надання абстракцій над інфраструктурою платформи виконання;
— модулі, що надають функціональність для обробки даних та представлення у вигляді окремих редюсерів.
— модуль для взаємодії з REST API;
— модуль, що містить реалізації конкретних компонентів та їх відображення.
— Модуль який відповідає за маршрутизацією.

В ході дослідження обрано оптимальне рішення для побудови архітектури системи, та використання і комбінування паттернів для отримання максимального результату. Для отримання максимального відклику та швидкодії було приміненно знання у ході дослідження управління пам’яттю в JavaScript, та реалізація управління у фреймворку React.

Розробленно унікальний графічний інтерфейс, на результатах аналізу інших подібних систем.

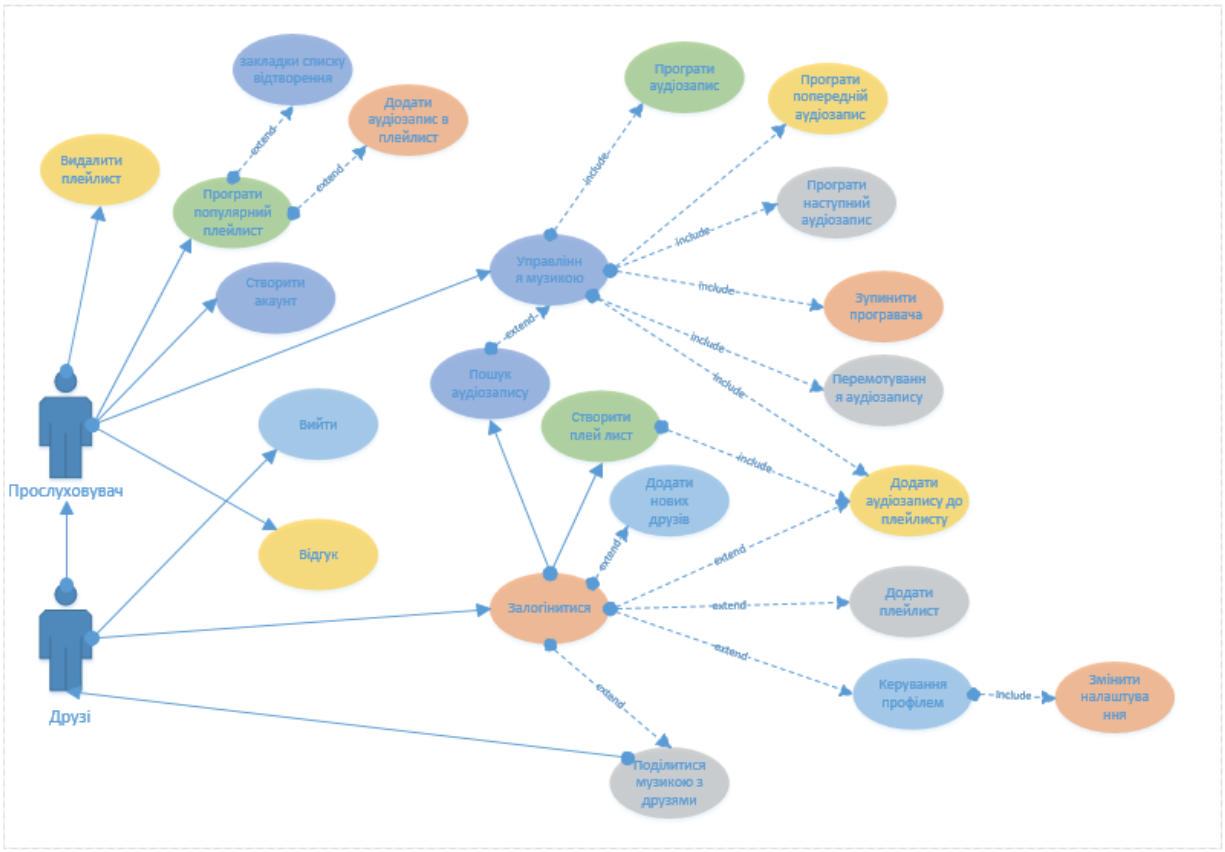
ПЕРЕЛІК ПОСИЛАНЬ

1. SPA Frameworks - a comparison between Angular, React and Vue [Електронний ресурс] – Режим доступу до ресурсу: https://www.snipclip.com/blog/spa-frameworks-a-comparison-between-angular-react-and-vue/?lang=en.
2. Интерфейс пользователя программного продукта [Електронний ресурс] – Режим доступу до ресурсу: https://gigabaza.ru/doc/27123.html.
3. 7 Steps of Effective Software Product Development Life Cycle [Електронний ресурс] – Режим доступу до ресурсу: https://relevant.software/blog/7-steps-for-effective-software-product-development
4. SOFTWARE DESIGN SPECIFICATION FOR A ONE RUNWAY AIRPORT/AIR TRAFFIC CONTROLLER SIMULATION, 2001. – 26 с.
5. CHRISTOPHER J FOX. Why Writing Software Design Documents Matters [Електронний ресурс] / CHRISTOPHER J FOX – Режим доступу до ресурсу: https://www.toptal.com/freelance/why-design-documents-matter.
6. Подробный обзор React Fiber [Електронний ресурс] – Режим доступу до ресурсу: https://uncleseneca.medium.com/80-react-fiber-66485d12bd37.
7. Karthik Kalyanaraman. A deep dive into React Fiber internals [Електронний ресурс] / Karthik Kalyanaraman – Режим доступу до ресурсу: https://blog.logrocket.com/deep-dive-into-react-fiber-internals/.
8. Inside Fiber [Електронний ресурс] – Режим доступу до ресурсу: https://indepth.dev/posts/1008/inside-fiber-in-depth-overview-of-the-new-reconciliation-algorithm-in-react.
9. Музыкальные стриминг-сервисы: рейтинг лучших, май 2021 [Електронний ресурс] – Режим доступу до ресурсу: https://ax.digital/music-streaming-tools/.
10. Обзор сервисов для прослушивания музыки [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/ru/post/532834/.

11. Flux: Архитектура приложений на React.js — всестороннее исследование [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/@marina.kovalyova/flux-the-react-js-application-architecture-773f515d068d.
12. Understanding passport.js authentication flow [Електронний ресурс] – Режим доступу до ресурсу: http://toon.io/understanding-passportjs-authentication-flow/.
13. Angular vs React vs Vue 2021 [Електронний ресурс] – Режим доступу до ресурсу: https://athemes.com/guides/angular-vs-react-vs-vue/.
14. Что Такое React и Как Он Работает на Самом Деле? [Електронний ресурс] – Режим доступу до ресурсу: https://www.hostinger.com.ua/rukovodstva/chto-takoe-react.
15. Библиотека React: особенности, перспективы, ситуация на рынке труда [Електронний ресурс] – Режим доступу до ресурсу: https://ru.hexlet.io/blog/posts/biblioteka-react-review-article.
16. Под капотом у React. Пишем свою реализацию с нуля [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/ru/post/458916/.
17. Что такое SPA или одностраничный портал [Електронний ресурс] – Режим доступу до ресурсу: http://www.codenet.ru/webmast/js/spa/.
18. ВЕБ ПРИЛОЖЕНИЕ. РАЗНИЦА МЕЖДУ САЙТОМ, ВЕБ-ПРИЛОЖЕНИЕМ, SPA И PWA [Електронний ресурс] – Режим доступу до ресурсу: https://webcase.com.ua/blog/cho-takoe-web-prilozhenie-vse-vidy/.
19. Single-page application vs. multiple-page application [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58.
20. Review: MongoDB takes on the world [Електронний ресурс] – Режим доступу до ресурсу: https://www.infoworld.com/article/3300619/review-mongodb-takes-on-the-world.html.
21. Flux [Електронний ресурс] – Режим доступу до ресурсу: https://facebook.github.io/flux/docs/overview.

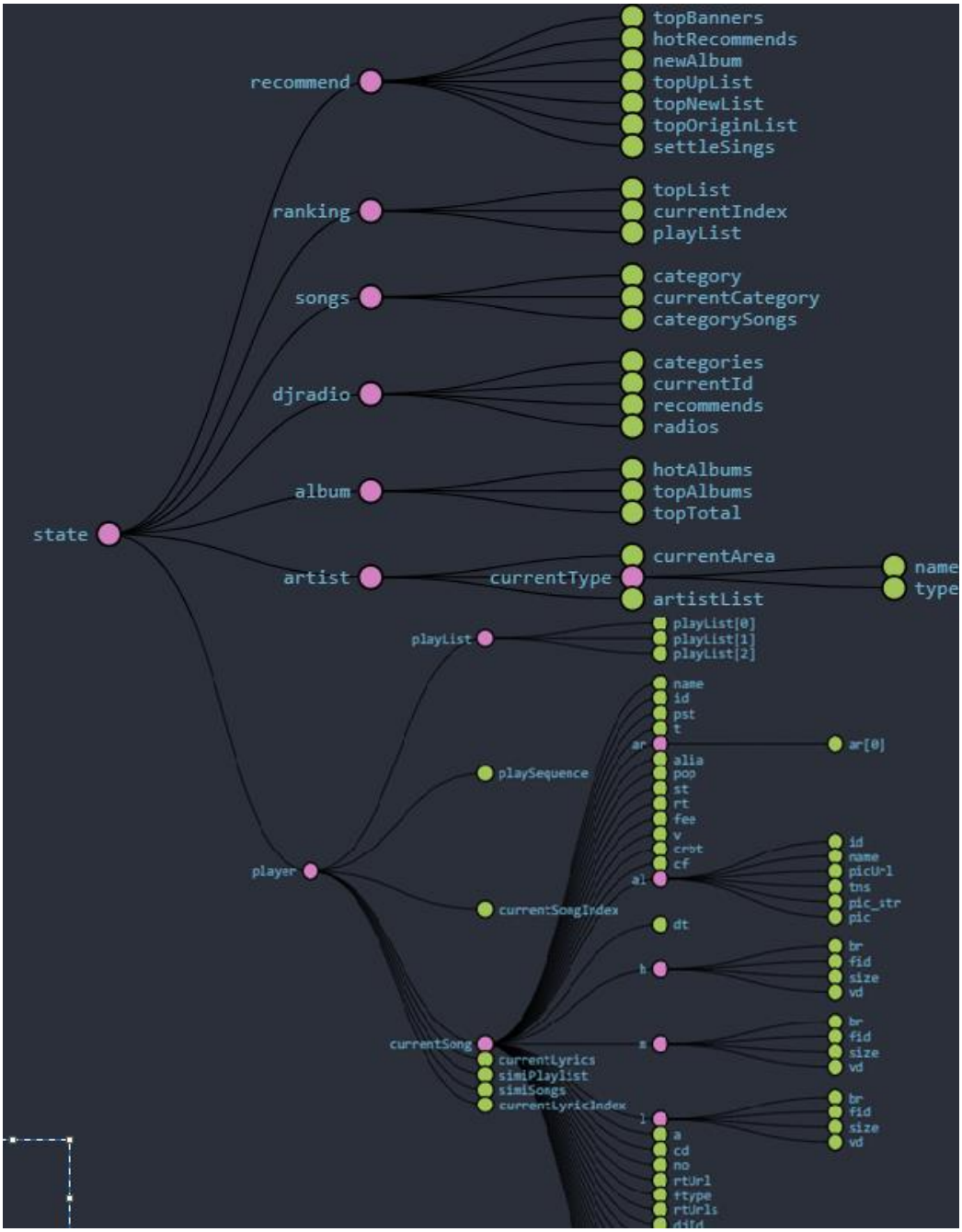
ДОДАТОК А

Діаграма сценарія використання користувача в системі



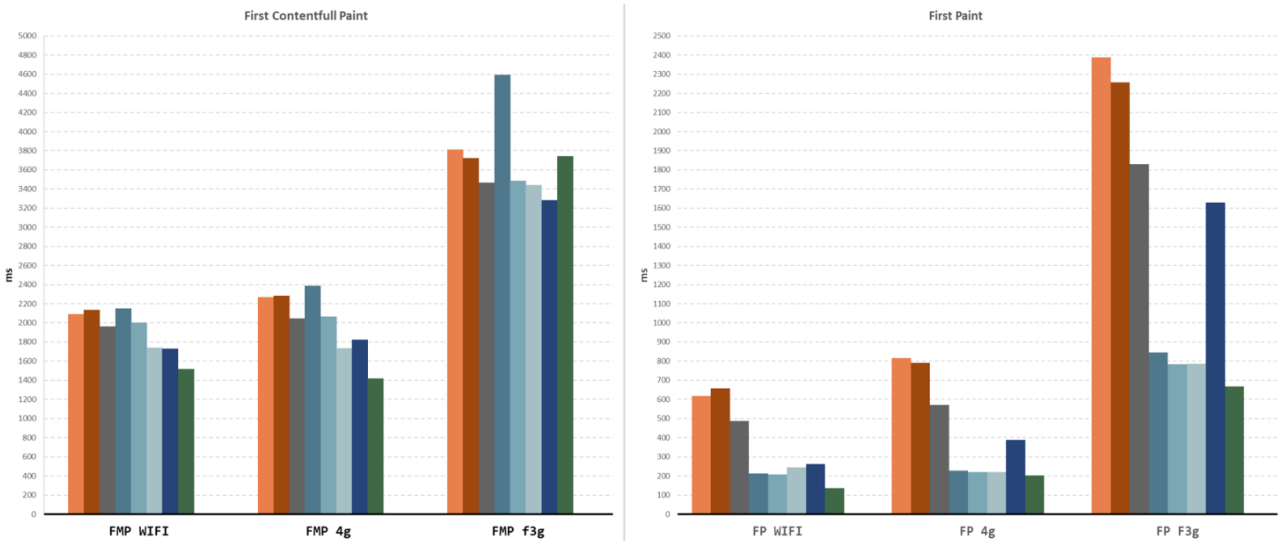
ДОДАТОК Б

Деревовидна структура головного контейнера стану



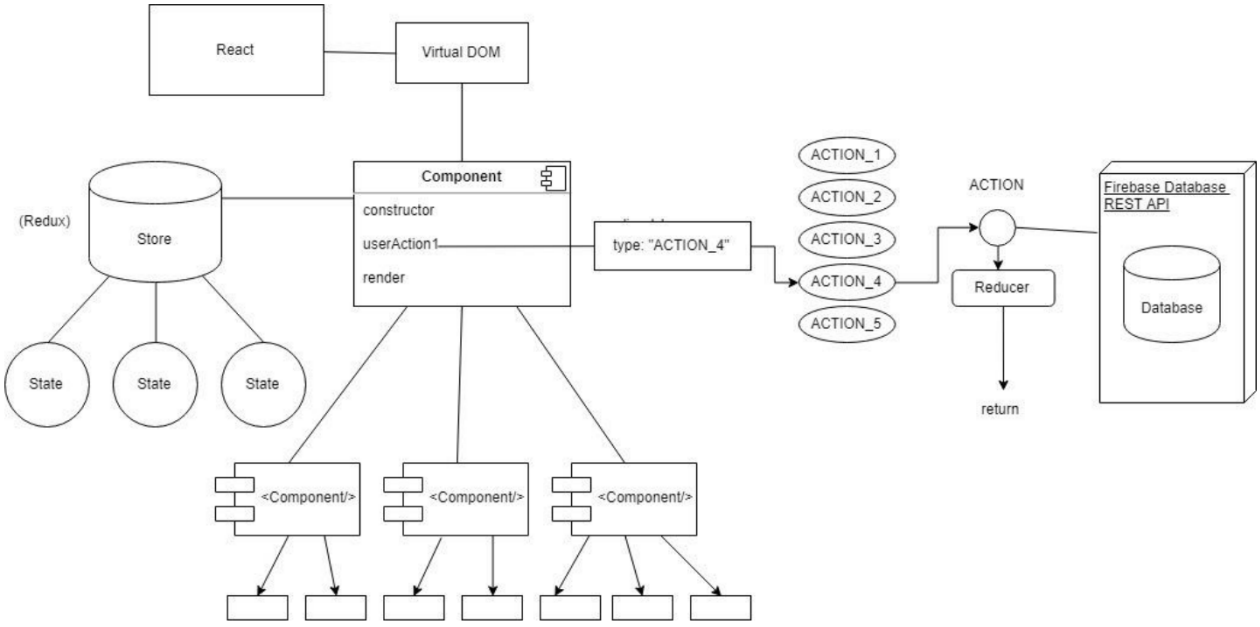
ДОДАТОК В

Таблиця швидкості загрузки системи в браузері



ДОДАТОК Г

Схема архітектури односторінкового застосунку



ДОДАТОК Ґ

Лістинг коду

```
export default combineReducers({
  recommend: recommendReducer,
  top: topReduc,
  songs: songsReduc,
  radio: radioReduc,
  albums: albumsReduc,
  artists: artistReduc,
  player: playReduc
})

export  memo(function AppPlay() {
  const [isPlaying, setIsPlaying] = useState(false);
  const [duration, setDuration] = useState(0);
  const [currentTime, setCurrentTime] = useState(0);
  const [progress, setProgress] = useState(0);
  const [isChanging, setIsChanging] = useState(false);
  const [showPanel, setShowPanel] = useState(false);
  const {
    currentSong,
    currentLyrics,
    currentLyricIndex,
    playList,
    playSequence
  } = useSelector(state => ({
```

```
  currentSong: state.getIn(["player", "currentSong"]),
  currentLyrics: state.getIn(["player", "currentLyrics"]),
  currentLyricIndex: state.getIn(["player", "currentLyricIndex"]),
  playList: state.getIn(["player", "playList"]),
  playSequence: state.getIn(["player", "playSequence"])
}), shallowEqual);
const dispatch = useDispatch();
const audioRef = useRef();
useEffect(() => {
  dispatch(getSongDetailAction(120));
}, [dispatch]);
useEffect(() => {
  audio.current.src = getPlayUrl(currentSong.id);
  audio.current.play().then(res => {
    setIsPlaying(true);
  }).catch(err => {
    setIsPlaying(false);
  });
  setDuration(currentSong.dt);
}, [currentSong]);
const play = useCallback(() => {
  setIsPlaying(!isPlaying);
  isPlaying ? audio.current.pause() : audioRef.current.play().catch(err => {
    setIsPlaying(false);
  });
}, [isPlaying]);
const timeUpdate = (e) => {
  const currentTime = e.target.currentTime;
  if (!isChanging) {
```

```
    setCurrentTime(currentTime);
    setProgress((currentTime * 1000) / duration * 100);
  }
  let Length = currentLyrics.length;
  let i = 0;
  for (; i < lrcLength; i++) {
    const lrcTime = currentLyrics[i].time;
    if (currentTime * 1000 < lrcTime) {
      break
    }
  }
  const finalIndex = i - 1;
  if (finalIndex !== currentLyricIndex) {
    dispatch(changeCurrentLyricIndexAction(finalIndex));
    message.open({
      content: currentLyrics[finalIndex].content,
      key: "lyr",
      duration: 0,
      className: 'message',
    })
  }
}
const timeEnded = () => {
  if (playSequence === 2 || playList.length === 1) {
    audioRef.current.currentTime = 0;
    audioRef.current.play();
  } else {
    dispatch(changePlaySongAction(1));
  }
```

```
import axios from 'axios';

const instance = axios.create({
   baseURL:'https://bloggy-api.herokuapp.com'
});

export const ett = async() =>{
   const {data} = await instance.get('/posts');
   return data;
}
export const delete = async(postId) =>{
   const { data} = await instance.delete(`/posts/${postId}`);
   return data;
}
export const getSingle = async(postId) =>{
   const {data} = await instance.get(`posts/${postId}?_embed=comments`);
   return data;
}
export const createNew = async({title,body}) =>{
   const {data} = await instance.post(`posts`,{
      "title": title,
      "body": body
      }
   );
   return data;
}
export const createNewComments = async({postId,body}) =>{
   const {data} = await instance.post(`/comments`,{
      "postId": postId,
      "body": body
      }
   );
   return data;
}
export const apdaite = async({postId,title,body}) =>{
   const {data} = await instance.put(`posts/${postId}`,{
      "title": title,
      "body": body
      }
   );
```

```
    return data;
}

  }
  const slider = useCallback((value) => {
    setProgress(value);
    const time = value / 100.0 * duration / 1000;
    setCurrentTime(time);
    setIsChanging(true);
  }, [duration])
  const sliderAfter = useCallback((value) => {
    const time = value / 100.0 * duration / 1000;
    audioRef.current.currentTime = time;
    setCurrentTime(time);
    setIsChanging(false);
    if (!isPlaying) {
      play();
    }
  }, [duration, isPlaying, play]);
memo(function Panel() {
  const { currentLyrics, currentLyricIndex } = useSelector(state => ({
    currentLyrics: state.getIn(["player", "currentLyrics"]),
    currentLyricIndex: state.getIn(["player", "currentLyricIndex"])
  }), shallowEqual);
  const panel= useRef();
  useEffect(() => {
    if (currenIndex > 0 && currentIndex < 3) return;
    scrollTo(panel.current, (currentIndex - 3) * 12)
  }, [currentIndex])
export default memo(function HYPlayHeader() {
```

```
  const { playList, currentSong } = useSelector(state => ({
    playList: state.getIn(["player", "playList"]),
    currentSong: state.getIn(["player", "currentSong"])
  }), shallowEqual);
export  memo(function PlayList() {
  const { playList, currentSongIndex } = useSelector(state => ({
    playList: state.getIn(["player", "playList"]),
    currentSongIndex: state.getIn(["player", "currentSongIndex"])
  }), shallowEqual);
const SongAction = (song) => ({
  type: actionTypes.CHANGE_SONG,
  song
});

const cSongIndexAction = (index) => ({
  type: actionTypes.CHANGE_SONG_INDEX,
  index
});

const ListAction = (playList) => ({
  type: actionTypes.CHANGE_LIST,
})



export const changePlaySongAction = (tag) => {
  return (dispatch, getState) => {
    let currentSongIndex = getState().getIn(["player", "currentSongIndex"]);
    const playSequence = getState().getIn(["player", "playSequence"]);
    const playList = getState().getIn(["player", "playList"]);
```

```
    switch (playSequence) {
      case 1:
        currentSongIndex = Math.floor(Math.random() * playList.length);
        break;
      default:
        currentSongIndex += tag;
        if (currentSongIndex === playList.length) currentSongIndex = 0;
        if (currentSongIndex === -1) currentSongIndex = playList.length - 1;
    }
    const currentSong = playList[currentSongIndex];
    dispatch(changeCurrentSongIndexAction(currentSongIndex));
    dispatch(changeCurrentSongAction(currentSong));
    getLyric(currentSong.id).then(res => {
      const lrcString = res.lrc.lyric;
      const lyrics = parseLyric(lrcString);
      dispatch(changeLyricsAction(lyrics));
    });
  }
}



    const playList = getState().getIn(["player", "playList"]);

    const songIndex = playList.findIndex(song => song.id === ids);
    if (songIndex !== -1) {
      const currentSong = playList[songIndex];
      dispatch(changeCurrentSongIndexAction(songIndex));
      dispatch(changeCurrentSongAction(currentSong));
```

```
    } else {
      getSongDetail(ids).then(res => {
        const song = res.songs && res.songs[0];
        if (!song) return;

        const newPlayList = [...playList];
        newPlayList.push(song);
        dispatch(changePlayListAction(newPlayList));

        dispatch(changeCurrentSongIndexAction(newPlayList.length - 1));
        dispatch(changeCurrentSongAction(song));
      });
    }


    Lyric(ids).then(res => {
      const String = res.lrc.lyric;
      const lyrics = parseLyric(lrcString);
      dispatch(cLyricsAction(lyrics));
    });
  }
}

export const Action = () => {
  return (dispatch, getState) => {
    const id = getState().getIn(["player", "curntSong"]).id;
    if (!id) return;

    Playlist(id).then(res => {
```

```
      dispatch(Action(res));
    })
  }
}

export const Action = () => {
  return (dispatch, getState) => {
    const id = getState().getIn(["player", "currentSong"]).id;
    if (!id) return;

    Song(id).then(res => {
      dispatch(SongsAction(res));
    })
  }
}

ArtSchema.statics.addSong = (artistId, songId) => {
  const Artist = mongoose.model("artists");
  const Song = mongoose.model("songs");
  return Artist.findById(artistId).then(artist => {
    return Song.findById(songId).then(song => {
      artist.songs.push(song); // are these push ?
      return Promise.all([artist.save(), song.save()]).then(
        ([artist, song]) => artist
      )
    })
  })
}
import { createSlice,createAsyncThunk, isRejectedWithValue } from '@reduxjs/tool
kit';
```

```
import {getPost,deletePost,getSinglePost,createNewPost,createNewComments,apdai
tePost} from '../../API/postAPI';

const initialState = {
    posts:[],
    loading:'idle',
    error:null,
    currentRequestId: "",
  };

  export const fetchPosts = createAsyncThunk('posts/fetchPosts', async (
    _,{rejectWithValue}
  ) => {
    try {
      const list = await getPost();
      return list;
    } catch (err) {
      return rejectWithValue([], err);
    }
  })
  export const sentNewPost = createAsyncThunk('posts/sentNewPost', async (
    data,{rejectWithValue}
  ) => {
    try {
      const list = await createNewPost(data);
      return list;
    } catch (err) {
      return rejectWithValue([], err);
    }
  })
  export const sentNewPostComment = createAsyncThunk('posts/sentNewPostCom
ment', async (
    data,{rejectWithValue}
  ) => {
    try {
      const list = await createNewComments(data);
      const listLike = await getSinglePost(data.postId);
      return list;
    } catch (err) {
      return rejectWithValue([], err);
```

```
    }
  })
 export const DeleteSinglePost = createAsyncThunk('posts/fetchDeletePosts', async (
    postId,{rejectWithValue}
  ) => {
    try {
      const list = await deletePost(postId);
      const listLike = await getPost();
      return list;
    } catch (err) {
      return rejectWithValue([], err);
    }
  })
export const getSinglePosts = createAsyncThunk('posts/getSinglePosts', async (
    postId,{rejectWithValue}
  ) => {
    try {
      const list = await getSinglePost(postId);
      return list;
    } catch (err) {
      return rejectWithValue([], err);
    }
  })
export const apdaiteThePost = createAsyncThunk('posts/apdaiteThePost ', async (
    data,{rejectWithValue}
  ) => {
    try {
      const list = await apdaitePost(data);
      //const listLike = await getPost();
      return list;
    } catch (err) {
      return rejectWithValue([], err);
    }
  })

export const postSlice =  createSlice({
    name :'posts',
    initialState,
    reducers:{
      increment(state) {
```

```
    state.value++
    state.loading = 'pending'
  },
},
extraReducers :{
    [fetchPosts.pending]: (state, { meta }) => {
      state.currentRequestId = meta;
      state.loading = "pending";
    },
    [fetchPosts.fulfilled]: (state, { meta, payload }) => {
      if (meta.requestId === state.currentRequestId.requestId) {
        state.posts = state.posts.concat(payload)
        state.loading = "fin";
        state.currentRequestId = "";
      }
    },
    [fetchPosts.rejected]: (state, { meta, payload,error }) => {
      if (state.currentRequestId === meta) {
        state.currentRequestId = meta;
        state.loading = "fin";
        state.posts = payload;
        state.error = error;
      }
    },
    [getSinglePosts.fulfilled]: (state, { meta, payload }) => {
      if (meta.requestId === state.currentRequestId.requestId) {
        state.posts = payload;
        state.loading = "fin";
        state.currentRequestId = "";
      }
    },
    [getSinglePosts.pending]: (state, { meta }) => {
      state.currentRequestId = meta;
      state.loading = "pending";
    },
    [getSinglePosts.rejected]: (state, { meta, payload, error }) => {
      if (state.currentRequestId === meta) {
        state.currentRequestId = meta;
        state.loading = "fin";
        state.posts = payload;
```

```
    state.error = error;
  }
},

[DeleteSinglePost.fulfilled]: (state, { meta, payload }) => {
  if (meta.requestId === state.currentRequestId.requestId) {
    state.posts = [];
    state.loading = "fin";
    state.currentRequestId = "";
  }
},
[DeleteSinglePost.pending]: (state, { meta }) => {
  state.currentRequestId = meta;
  state.loading = "pending";
},
[DeleteSinglePost.rejected]: (state, { meta, payload, error }) => {
  if (state.currentRequestId === meta) {
    state.currentRequestId = meta;
    state.loading = "fin";
    state.posts = payload;
    state.error = error;
  }
},
[sentNewPost.fulfilled]: (state, { meta, payload }) => {
  if (meta.requestId === state.currentRequestId.requestId) {
    state.posts.push(payload)
    state.loading = "fin";
    state.currentRequestId = "";
  }
},
[sentNewPost.pending]: (state, { meta }) => {
  state.currentRequestId = meta;
  state.loading = "pending";
},
[sentNew.rejected]: (state, { meta, payload, error }) => {
  if (state.currentRequestId === meta) {
    state.currentRequestId = meta;
    state.loading = "fin";
    state.posts = payload;
    state.error = error;
```

```
  }
},
[sentNewComment.fulfilled]: (state, { meta, payload }) => {
  if (meta.requestId === state.currentRequestId.requestId) {
    //state.posts.push(payload)
    state.loading = "fin";
    state.currentRequestId = "";
  }
},
[sentNewComment.pending]: (state, { meta }) => {
  state.currentRequestId = meta;
  state.loading = "pending";
},
[sentNewComment.rejected]: (state, { meta, payload, error }) => {
  if (state.currentRequestId === meta) {
    state.currentRequestId = meta;
    state.loading = "fin";
    state.posts = payload;
    state.error = error;
  }
},
  [apdaiteThe.fulfilled]: (state, { meta, payload }) => {
  if (meta.requestId === state.currentRequestId.requestId) {
    //
    state.loading = "fin";
    state.currentRequestId = "";
  }
},
[apdaiteThePost.pending]: (state, { meta }) => {
  state.currentRequestId = meta;
  state.loading = "pending";
},
[apdaiteThePost.rejected]: (state, { meta, payload, error }) => {
  if (state.currentRequestId === meta) {
    state.currentRequestId = meta;
    state.loading = "fin";
    state.posts = payload;
    state.error = error;
  }
},
```

```
    }
});

export const {postsReceived,postsLoading} = postSlice.actions;

export default postSlice.reducer;

ArtistSchema.statics.addGenre = (artistId, genreId) => {
  const Artist = mongoose.model("artists");
  const Genre = mongoose.model("genres");
  return Artist.findById(artistId).then(artist => {
    return Genre.findById(genreId).then(genre => {
      artist.genres.push(genre);
      genre.artists.push(artist);
      return Promise.all([artist.save(), genre.save()]).then(
        ([artist, genre]) => artist
      )
    })
  })
}

const PlaylitSchema = new Schema({
  user: {
    type: Schema.Types.ObjectId,
    ref: 'users'
  },
  title: {
    type: String,
    required: true
  },
  description: {
```

```
    type: String,
    required: false
  },
  songs: [{
    type: Schema.Types.ObjectId,
    ref: 'songs'
  }]
});

PlaylistSchema.statics.findSongs = (songId) => {
  return this.findById(songId)
    .populate('songs')
    .then(playList => playList.songs);
}

PlaylistSchema.statics.addSong = (playlistId, songId) => {
  const Playlist = mongoose.model("playlists");
  const Song = mongoose.model("songs");

  return Playlist.findById(playlistId).then(playlist => {
    return Song.findById(songId).then(song => {
      playlist.songs.push(song);
      song.playlists.push(playlist);

      return Promise.all([playlist.save(), song.save()]).then(
        ([playlist, song]) => playlist
      );
    });
  });
```

```
};

let SongSchema = new Schema({
  title: {
    type: String,
    required: true
  },
  artist: {
    type: Schema.Types.ObjectId,
    ref: "artists"
  },
  imageUrl: {
    type: String,
    required: true
  },
  songUrl: {
    type: String,
    required: true
  },
  playlists: [
    {
      type: Schema.Types.ObjectId,
      ref: "playlists"
    }
  ],
  likes: [{
    type: Schema.Types.ObjectId,
    ref: "users"
  }]
```

```
});

SonSchema.statics.addLike = (songId, userId) => {
  const Song = mongoose.model("songs");
  const User = mongoose.model("users");
  return Promise.all([Song.findById(songId), User.findById(userId)])
    .then(([song, user]) => {
      song.likes.push(user);
      user.likeSongs.push(song);
      return Promise.all([song.save(), user.save()])
        .then(([song, user]) => song);
    })
}

SongSchema.statis.removeLike = (songId, userId) => {
  const Song = mongoose.model("songs");
  const User = mongoose.model("users");
  return Promise.all([Song.findById(songId), User.findById(userId)])
    .then(([song, user]) => {
      song.likes.pull(user);
      user.likedSongs.pull(song);
      return Promise.all([song.save(), user.save()])
        .then(([song, user]) => song);
    })
}

const mutation = new GraphQLObjectType({
  name: "Mutation",
  fields: {
```

```
newPlaylist: {
  type: PlaylistType,
  args: {
    title: { type: GraphQLString },
    description: { type: GraphQLString },
    user: { type: GraphQLID },
  },
  resolve(_, { title, description, user }) {
    return new Playlist({ title, description, user }).save();
  }
},
deletePlaylist: {
  type: PlaylistType,
  args: { id: { type: GraphQLID } },
  type: PlaylistType,
  args: {
    playlistId: { type: GraphQLID },
    songId: { type: GraphQLID }
  },
  resolve(_, { playlistId, songId }) {
    return Playlist.addSong(playlistId, songId);
  }
},
removePlaylistSong: {
  type: PlaylistType,
  args: {
    playlistId: { type: GraphQLID },
    songId: { type: GraphQLID }
  },
```

```
  resolve(_, { playlistId, songId }) {
    return Playlist.removeSong(playlistId, songId);
  }
},
register: {
  type: UserType,
  args: {
    name: { type: GraphQLString },
    email: { type: GraphQLString },
    password: { type: GraphQLString }
  },
  resolve(_, args) {
    return AuthService.register(args);
  }
},
login: {
  type: UserType,
  args: {
    email: { type: GraphQLString },
    password: { type: GraphQLString }
  },
  resolve(_, args) {
    return Iervice.login(args);
  }
},
logout: {
  type: UserType,
  args: {
    _id: { type: GraphQLID }
```

```
  },
  resolve(_, args) {
    return AuthService.logout(args);
  }
},
verUser: {
  type: Type,
  args: {
    token: { type: GraphQLString }
  },
  resolve(_, args) {
    return AuthService.verifyUser(args);
  }
},

Artist: {
  type: Type,
  args: {
    name: { type: GraphQLString },
    imageUrl: { type: GraphQLString }
  },
  resolve(parentValue, { name, imageUrl }) {
    return new Artist({ name, imageUrl }).save();
  }
},
Song: {
  type: Type,
  args: {
    title: { type: GraphQLString },
```

```
    artist: { type: GraphQLID },
  },
  resolve(parentValue, { title, artist, imageUrl, songUrl }) {
    return new Song({ title, artist, imageUrl, songUrl }).save();
  }
},
Genre: {
  type: Type,
  args: {
    name: { type: GraphQLString },
    imageUrl: { type: GraphQLString }
  },
  resolve(parentValue, { name, imageUrl }) {
    return new Genre({ name, imageUrl }).save();
  }
},

addArtistSong: {
  type: ArtistType,
  args: {
    artistId: { type: new GraphQLNonNull(GraphQLID) },
    songId: { type: new GraphQLNonNull(GraphQLID) }
  },
  resolve(parentValue, { artistId, songId }) {
    return Artist.addSong(artistId, songId);
  }
},
artistGenre: {
  type: ArstType,
```

```
    args: {
      artistId: { type: new GraphQLNonNull(GraphQLID) },
      genreId: { type: new GraphQLNonNull(GraphQLID) }
    },
    resolve(parentValue, { artistId, genreId }) {
      return Artist.addGenre(artistId, genreId);
    }
  },
  Song: {
    type: SongType,
    args: {
      songId: { type: new GraphQLNonNull(GraphQLID) },
      userId: { type: new GraphQLNonNull(GraphQLID) }
    },
    resolve(parentValue, { songId, userId }) {
      return Song.addLike(songId, userId)
    }
  },
    type: SongType,
    args: {
      songsId: { type: new GraphQLNonNull(GraphQLID) },
      usersId: { type: new GraphQLNonNull(GraphQLID) },
   recoId: { type: new GraphQLNonNull(GraphQLID) }
    },
    resolve(parentValue, { songId, userId }) {
      return Song.Like(songId, userId);
    }
  }
}
```